# Mascarpone und XML-CONCUR

## Ein Editor und ein Verarbeitungsmodell für multi-strukturierte Daten

Oliver Schonefeld
30. September 2005

Diplomarbeit im Studiengang Naturwissenschaftliche Informatik
Technische Fakultät, Universität Bielefeld

**Gutachter:**
Dr. Hans-Jürgen Eikmeyer
Prof. Dr. Franz Kummert
Dr. Andreas Witt

**Zusammenfassung**

Die vorliegende Arbeit beschreibt die Entwicklung eines Verarbeitungsmodells für multi-hierarchisch strukturierte Daten und dessen Implementierung als Prototyp eines Annotationswerkzeugs. In Anlehnung an die Syntax von SGML CONCUR wird eine Dokument-Syntax und ein entsprechendes Verarbeitungsmodell definiert, die es erlauben, Annotationen auf mehreren Ebenen in einem XML-ähnlichen Dokument vorzunehmen.

# Inhaltsverzeichnis

# Kapitel 1

# Einleitung

## 1.1 Einleitung

Die Extensible Markup Language (XML) hat sich in den letzen Jahren zu einem Standard beim Austausch und der Speicherung von Daten und Dokumenten etabliert. Auch in vielen wissenschaftlichen Bereichen wird sie vermehrt eingesetzt. Die Anwendungen erstrecken sich von der Definition von Formaten zum Speicherung und Austausch von Daten bis zur Annotation von Text-Dokumenten oder Korpora transkribierter Sprache. Ein Vorteil von XML gegenüber anderen Formaten ist seine Vielseitigkeit, denn es lassen sich (fast) beliebige Strukturen in einem XML-Dokument speichern. Diese Tastsache hat stark zu seiner Verbreitung beigetragen.

Je nach Forschungsfragestellung gibt es oft unterschiedliche Sichtweisen auf die Daten. So ist eine unterschiedliche Auszeichnung und Analyse der Daten erforderlich. Die Sichtweise hängt vom wissenschaftlichen Standpunkt des Betrachters und der Arbeitsweise ab. Auch spielen dabei Überlegenungen zu weiteren Verarbeitung der Daten eine Rolle.

Oft gibt es verschiedene Sichtweisen auf die gleichen Daten und so ist es durchaus üblich, die gleichen Daten mehrfach zu annotieren. Es entstehen dabei verschiedene Dokumente, die durch unterschiedliche XML-Schemata annotiert und strukturiert sind. Diese Annotation wird oft auch von mehreren verschiedenen Personen unabhängig voneinander vorgenommen. Bei der weiteren Analyse ist es jedoch interessant, diese unterschiedlichen Dokumente wieder zusamenzuführen, um die verschiedenen Sichtweisen in Beziehung zueinander zu setzen.

Die Zusammenführung der verschiedenen Dokumente ist jedoch nicht ganz unproblematisch. Die unterschiedlichen Annotationen können überlappende Strukturen bilden und diese können nicht ohne weiteres in XML abgebildet werden. Die Struktur eines XML-Dokuments muss eine Baumstruktur ergeben und darf keine Überlappungen enthalten.

Mirco Hilbert definiert in [11] eine Dokumenten-Syntax, und ein Verarbeitungsmodell, das die Annotation von mehrfach-strukturierten Daten erlaubt. Die Zielsetzung dieser Arbeit ist, auf den Ergebnissen von Hilbert aufzubauend, das Dokument-Format, das XML-CONCUR (XMC) getauft wurde um einige Elemente zu erweitern und ein alternatives Verarbeitungsmodell zu definier. Das Verarbeitungsmodell und ein Annotationswerkzeug sollen implementiert werden.

## 1.2 Gliederung

Im Folgenden werden einige grundlegenden Begriffe geklärt. Ein prinzipielles Verständnis vom XML und Auszeichnungssprachen wird dabei vorausgesetzt.

Im zweiten Abschnitt folgt eine Einführung in die Problematik der multihierarchisch strukturierten Daten und es werden einige alternative Ansätze, wie dieses Problem angegangen werden kann, beleuchtet. Die Vor- wie auch die Nachteile der verschiedenen Lösungen werden dabei betrachtet.

Das dritte Kapitel widmet sich dem Entwurf einer Dokument-Syntax für multi-hierarchisch strukturierte Dokumente. Der Vorschlag von Hilbert wird in diesem Kapitel vorgestellt und durch weitere Elemente erweitert.

Im vieren Abschnitt werden die beiden XML-CONCUR-Verarbeitungsmodelle vorgestellt. Neben dem Modell von Hilbert wird in dieser Arbeit ein weiterer Ansatz für ein Verarbeitungsmodell definiert. Am Ende des Kapitels werden beide Modelle miteinander verglichen und die Vor- und Nachteile des jeweiligen Modells aufgeführt.

Das fünfte Kapitel widmet sich der Implementierung des in dieser Arbeit definierten Verarbeitungsmodells und des Annotationswerkzeugs.

## 1.3 Begrifflichkeiten

### 1.3.1 Datenmodell

Ein *Datenmodell* beschreibt die logische Struktur von Daten. Dies kann beispielsweise eine Baum-, Tupel- oder Listenstruktur oder auch eine Kombination dieser Strukturen sein.

Durch das Datenmodell wird ausschliesslich ein abstraker Datentyp definiert, der durch eine Implementierung konkretisiert werden muss. Diese Implementierung realisiert dann sowohl die eigentliche Datenrepräsentation als auch ein Application Programming Interface (API), das Zugriffsmethoden auf die Datenstrukturen zur Verfügung stellt.

Die eigentlichen *Daten* sind im Kontext eines Datenmodells atomare Einheiten, die nicht weiter strukturiert werden können. Je nach Anwendung kann

es sich um einen Zeichenvorrat, beispielsweise UTF-8, oder Binärdaten wie Bilder handeln.

### 1.3.2 Dokument-Syntax

Eine *Dokument-Syntax* legt die Serialisierung der Datenstruktur eines vorgegeben Datenmodells fest. Der Begriff *Dokument* bezeichnet dabei eine Instanz des Datenmodells. Meist entspricht ein Dokument einer Datei; es kann jedoch auch aus mehreren Dateien bestehen.

Das Ziel einer Dokument-Syntax ist es, ein Dokument durch Hinzufügen von *Strukturinfomationen* derart zu gestalten, dass die ursprüngliche Datenmodell-Instanz durch *parsen* des Dokuments wiederhergestellt werden kann. Ein Dokument besteht aus *Primärdaten* und den eingefügten Stukturinformationen. Im Falle von XML werden die Strukturinformationen in spitzen Klammern ("`<...>`") eingeschlossen, um sie von den Primärdaten zu trennen.

Das Einfügen von Strukturinformationen in die Primärdaten wird als *Annotation* bezeichnet; die Realisierung der Stukturinformationen werden in der XML/SGML-Welt als *Markup* oder *Tags* bezeichnet.

Ein *Annotationsschema* definiert eine Menge von Regeln, die das Vokabular der Strukturinformationen und deren syntaktische Beziehungen zueinander festlegen. In einer *Annotations-Schemasprache* können diese Regeln notiert werden. Für XML sind zur Zeit drei Schemasprachen gebräuchlich: Document Type Definitions (DTDs), XML-Schema und RelaxNG. Wenn ein Dokument den Regeln einer Schemasprache entspricht, wird es als *valide* bezeichnet.

### 1.3.3 Verarbeitungsmodell

Hilbert definiert in [11] ein *Verarbeitungsmodell* als eine Sicht auf das Datenmodell, die von der gewählten Dokument-Syntax abhängig ist. Im mathematischen Sinne sei ein Verarbeitungsmodell eine Abbildung oder Transformation eines Datenmodells. Ein Verarbeitungsmodell entspricht einem speziellen Datenmodell mit der Zielsetzung der Verarbeitung der durch das Datenmodell strukturierten Daten.

Ein Verarbeitungsmodell muss sich nicht zwangsläufig vom Datenmodell unterscheiden und es kann durchaus mehrere Verarbeitungsmodell zu einem Datenmodell geben. Für XML beispielsweise gibt es mit dem Document Objekt Model (DOM) und dem XML Information Set zwei Datenmodelle.

# Kapitel 2

# Multi-hierarchische Daten und Concurrent Markup

## 2.1 Multi-hierarchisch strukturierte Daten

Bei der Weiterverarbeitung und Analyse von Daten werden oft verschiedene Sichtweisen auf diese Daten benötigt, da unterschiedliche Analyseziele erreicht werden sollen. Je nach Interesse oder Fragestellung stehen andere Aspekte im Mittelpunkt. Eine spezifische Sicht auf die Daten und damit verbundene (Meta-)Informationen bezeichnet Hilbert in [11] als *Informationsebene*.

Bei der lingustischen Annotation von prosaischen oder poetischen Texten ergeben sich verschiedene Informationsebenen. Für ein Gedicht ließe sich zum einen die lyrische Struktur, also Strophen und Verse, untersuchen, zum anderen könnte man die linguistische Struktur, also beispielsweise Sätze, Wörter und Morpheme betrachten. Jede Informationsebene besitzt in diesen Fällen eine eigene *hierarchische Struktur*, da zum Beispiel Strophen aus Versen bestehen, und Sätze aus Wörtern. Sollte eine Ebene nur eine flache Struktur haben, kann durch Einfügen einer (künstlichen) übergeordneten Ebene eine hierarchische Struktur geschaffen werden.

Das Konzept der Informationsebene wird durch eine *Annotationsebene* realisiert. Eine oder mehrere Informationsebenen können durch eine Annotationsebene umgesetzt werden. Für ein Gedicht können beispielsweise die beiden oben genannten Informationsebenen durch zwei Annotationsebenen umgesetzt werden: eine für die lyrische und eine für die linguistische Annotation.

Wenn die verschiedenen Informationsebenen untereinander verglichen oder im Rahmen weiterer Analysen vereinigt werden sollen, ist es wichtig, daß die verschiedenen Annotationen die Eigenschaft der *Primärdaten-Identität* besitzen. Das bedeutet, daß man die gleichen Dokumente als Resultat erhält, wenn sämtliche Strukturinformationen der Annotation entfernt werden.

Leider lässt sich diese Eigenschaft nicht einfach garantieren. Beispielsweise

resultiert eine unterschiedliche Formatierung der Annotation meist schon in Dokumenten, die nicht mehr primärdaten-identisch sind. Ein Zeilenumbruch oder ein Leerzeichen, das nicht an derselben Stelle in allen Dokumenten vorhanden ist, verletzt das Prinzip der Primärdaten-Identität.

Wenn es sich bei diesen Unterschieden nur um sogenannte Whitespaces (also Leerzeichen, Tabs und Zeilenumbrüche) handelt, kann man diesem Problem mit Hilfe einer "Whitespace-Normalisierung" begegnen. Dieser Prozess versucht, die Whitespaces in den verschiedenen Annotationen durch Einfügungen oder Löschungen anzugleichen. Dies ist kein triviales Unterfangen, denn es gibt "informationstragende" und "nicht-informationstragende" Whitespaces. Ein Leerzeichen zwischen zwei Wörtern ist informationstragend, eines das nur zur Formatierung der Annotation genutzt wird, ist nicht-informationstragend. Eine Whitespace-Normalisierung muss diese Unterschiede berücksichtigen.

In einigen Fällen ist es so, dass eine Informationsebene eine andere Informationsebene einschließt. Beispielsweise ist es bei der gleichzeitigen Annotation der Wort- und Satzstruktur möglich, mehrere Informationsebenen in einer Annotationsebene zusammenzufassen, da verschiedene Informationsebenen eine Hierarchie bilden können. Die Wortebene ist immer in die Satzebene eingebettet.

Eine solche Hierarchie lässt sich jedoch nicht immer finden und oft werden schon bei der Annotation verschiedene Informationsebenen zusammengefasst. Dadurch verliert man eine klare Trennung der Ebenen, die jedoch für spätere Analysen durchaus relevant sein kann. So wird auch das spätere Einfügen von neuen Informationsebenen erschwert.

Um ein Dokument validieren zu können, wird ein Dokument-Schema benötigt. Daher sind nicht nur in den Dokumenten die einzelnen Ebenen vermischt, sondern auch im Dokument-Schema. Das Einfügen einer weiteren Ebene in das Schema kann dann durchaus zu einer anspruchsvollen Aufgabe werden.

Oft lässt sich jedoch keine entsprechende Hierarchie zwischen den Informationsebenen finden, denn es gibt Überlappungen zwischen ihnen. Bei der Annotation des Gedichts kann es durchaus Überlappungen zwischen den Versen und der Satzstuktur geben, da sich ein Satz über mehrere Zeilen erstrecken kann.

Für einige Anwendungsziele ist es notwendig, mehrere physikalische Strukturen eines Textes, also seine Aufteilung in Bände, Kapitel, Seiten, Spalten, Abschnitte usw, zu annotieren, zum Beispiel, wenn verschiedene Druckausgaben eines Textes verglichen werden sollen. Zwischen den einzelnen Elementen kann es durchaus wieder Überlappungen geben: ein Absatz kann sich zum Beispiel über Seitengrenzen hinweg erstrecken.

Das Problem multipler Hierarchien und überlappender Annotation, auch *Concurrent Markup* genannt, ist kein konstruiertes Problem, sondern begegnet einem in der Praxis recht häufig. Die TEI Guidelines [16] führen in Kapitel 31

weitere praxisnahe Beispiele auf.

Wie kann also dem Problem des Concurrent Markup begegnet werden? In den folgenden Abschnitten werden verschiedene Lösungsansätze für die Annotation von multiplen Hierarchien betrachtet. Das Gedicht "Der Panther" von Rainer Maria Rilke ([3], Abbildung 2.1) soll als Beispieltext dienen. Es werden zwei Aspekte betrachtet: zum einen die lyrische und zum anderen die linguistische Struktur. Bei SGML/XML basierten Ansätzen werden dazu die entsprechenden Tagsets verwendet, die in den TEI Guidelines ([16]) definiert wurden. Bei anderen wird zur besseren Illustration eine analoge Struktur verwendet, die den Tagsets der TEI nachempfunden worden ist.

Um die Beispiele nicht unnötig zu vergrößern, wird nur die erste Strophe betrachtet. Sie besteht aus zwei Sätzen, die sich über vier Zeilen erstrecken. So ergibt sich in jedem Satz eine Überlappung mit der Annotation der Zeilen.

```
Der Panther

Sein Blick ist vom Vorübergehn der Stäbe
so müd geworden, dass er nichts mehr hält.
Ihm ist, als ob es tausend Stäbe gäbe
und hinter tausend Stäben keine Welt.

Der weiche Gang geschmeidig starker Schritte,
der sich im allerkleinsten Kreise dreht,
ist wie ein Tanz von Kraft um eine Mitte,
in der betäubt ein großer Wille steht.

Nur manchmal schiebt der Vorhang der Pupille
sich lautlos auf -. Dann geht ein Bild hinein,
geht durch der Glieder angespannte Stille -
und hört im Herzen auf zu sein.
```

Abbildung 2.1: "Der Panther" von Rainer Maria Rilke

Es ist durchaus möglich das Gedicht XML-konform, d. h. ohne Überlappungen zu annotieren, wenn die Annotation des Satzes die der Zeile enthalten würde. Diese Verschachtelung der Annotation ist jedoch durch die TEI Dokument-Grammatik nicht zugelassen. Sie würde nur mehrere Zeilen innerhalb eines Satzes erlauben, nicht jedoch mehrere (Teil-)Sätze innerhalb einer Zeile.

Es gibt jedoch auch überlappende Strukturen, die nicht so annotiert werden können. Im folgenden Dialog sprechen "Peter" und "Paul" miteinander. "Paul" ergänzt einen Satz, den "Peter" begonnen hat. Wenn sowohl die Sprecher-Turns als auch die gesprochenen Sätze annotiert werden sollen, kommt es an

dieser Stelle zu einer Überlappung, die nicht durch eine andere Schachtelung der Annotation umgangen werden kann.

> Peter: Hey Paul! Would you give me
> Paul: the Hammer?

## 2.2 SGML-basierte Ansätze

```
1  <!SGML "ISO 8879:1986" FEATURES OTHER CONCUR YES 2 >
2  <!DOCTYPE lg SYSTEM "teivers2.dtd">
3  <!DOCTYPE text SYSTEM "teiana2.dtd">
4  <(lg)lg>
5    <(text)text>
6      <(lg)l>
7        <(text)s>Sein Blick ist vom Vorübergehn der Stäbe
8      </(lg)l>
9      <(lg)l>
10       so müd geworden, dass er nichts mehr hält.</(text)s>
11     </(lg)l>
12     <(lg)l>
13       <(text)s>Ihm ist, als ob es tausend Stäbe gäbe
14     </(lg)l>
15     <(lg)l>
16       und hinter tausend Stäben keine Welt.</(text)s>
17     </(lg)l>
18   </(text)text>
19 </(lg)lg>
```

Abbildung 2.2: Annotation mit SGML-CONCUR

In der SGML Norm [1] ist bereits eine Spezifikation für Markup mit mehreren Annotationsebenen vorhanden. Diese optionale Eigenschaft von SGML heisst CONCUR, das für "Concurrent Markup" steht. Durch Setzen der Option "CONCUR" in der SGML Deklaration auf den Wert "YES n" kann man sie aktivieren. So können bis zu $n$ DTDs in einem Dokument verwendet werden. Um die Elemente den einzelnen DTDs zuordnen zu können, erhalten diese eine Dokumenttyp-Spezifikation, die ihnen in runden Klammern vorangestellt wird.

In Abbildung 2.2 ist das Rilke Gedicht mit zwei Annotationsebenen unter Verwendung von SGML-CONCUR annotiert. In Zeile 1 wird in der SGML Deklaration die Option "CONCUR" mit zwei Annotationsebenen aktiviert. Die Ebenen werden durch die "DOCTYPE" Deklarationen der Zeilen 2 und

3 definiert. Vor den einzelnen Elementnamen ist in den runden Klammern entweder der Dokument-Typ "lg" für die lyrische oder "text" für linguistische Struktur angegeben.

Wenn ein Element in allen verwendeten DTDs spezifiziert ist, kann der Dokument-Typ auch weggelassen werden. Das Element wird dann von allen Annotationsebenen gemeinsam verwendet.

Die "CONCUR"-Eigenschaft wurde bei der Spezifikation von XML nicht übernommen. Es gibt so gut wie keinen SGML Prozessor, der CONCUR unterstützt. Selbst Charles Goldfarb, einer der Hauptentwickler von SGML, sprach sich gegen die Verwendung von CONCUR aus:

> I therefore recommend that CONCUR is not be used to create multiple logical views of a document, such as verse-oriented and speech-oriented views of poetry. (Goldfarb, [10])

Trotz der einfachen und ansprechenden Syntax spielt SGML-CONCUR, zumindest in der Praxis, keine Rolle.

## 2.3 XML-basierte Ansätze

Im folgenden werden verschiedene XML-basierte Ansätze zur Realisierung von verschiedenen Annotationsebenen vorgestellt. Die Ansätze stammen, soweit nicht anders erwähnt, aus den Vorschlägen im Kapitel "Multiple Hierarchies" der TEI Guidelines ([16]).

### 2.3.1 Separate Annotation

Durch separate Annotation eines Dokuments kann sehr leicht eine Annotation auf verschiedenen Ebenen vorgenommen werden. Das Ursprungsdokument wird dazu einfach kopiert und dann annotiert. Für jede Informationsebene wird eine Kopie des Dokuments benötigt.

Die Abbildungen 2.3 und 2.4 zeigen die resultierenden XML-Dokumente, wenn zwei Ebenen annotiert werden. Die beiden XML-Dokumente sind vollkommen unabhängig voneinander.

Diese Methode hat den Vorteil, dass die Informationsebenen weder im Dokument noch im Dokumenten-Schema vermischt sind. Für jede Ebene kann ein eigenes Schema verwendet werden. Es gibt eine klare Trennung zwischen den Informationsebenen. Weiterhin kann man ohne Probleme das gleiche Schema für verschiedene Ebenen nutzen und das Hinzufügen (oder Entfernen) von Ebenen ist ebenfalls sehr einfach.

Die Aufteilung der Annotationsebenen auf mehrere Dateien bringt jedoch einige Probleme mit sich. Die Forderung nach Primärdaten-Identität lässt sich

```
1  <?xml version="1.0" encoding="iso-8859-1"?>
2  <!DOCTYPE lg SYSTEM "teivers2.dtd">
3  <lg>
4    <l>Sein Blick ist vom Vorübergehn der Stäbe</l>
5    <l>so müd geworden, dass er nichts mehr hält.</l>
6    <l>Ihm ist, als ob es tausend Stäbe gäbe</l>
7    <l>und hinter tausend Stäben keine Welt.</l>
8  </lg>
```

Abbildung 2.3: Separate Annotation der Versstruktur

```
1  <?xml version="1.0" encoding="iso-8859-1"?>
2  <!DOCTYPE text SYSTEM "teiana2.dtd">
3  <text>
4    <s>Sein Blick ist vom Vorübergehn der Stäbe
5       so müd geworden, dass er nichts mehr hält.</s>
6    <s>Ihm ist, als ob es tausend Stäbe gäbe
7       und hinter tausend Stäben keine Welt.</s>
8  </text>
```

Abbildung 2.4: Separate Annotation der Satzebene

nur schwer umsetzen. Allein eine unterschiedliche Formatierung der Annotation verletzt diese Forderung, da an unterschiedlichen Stellen unterschiedliche Whitespaces auftreten können. Durch eine "Whitespace-Normalisierung"[1] kann man versuchen dies wieder zu korrigieren. Fehler beim Annotieren, zum Beispiel das Löschen oder Einfügen eines "Non-Whitespace"-Zeichens, lassen sich so jedoch nicht korrigieren.

Weiterhin führt die separate Annotation zu Redundanz. Abgesehen davon, dass die Daten mehrfach gespeichert werden müssen, muss eine Annotation eventuell auch mehrfach ausgeführt werden. Eine syntaktische und eine semantische Analyse könnten jeweils auf einer eigenen Wort-Ebene aufsetzen. Auf beiden Ebenen müssten also die Worte annotiert werden. Der Versuch, dieses Problem zu umgehen und mit dokumentübergreifenden Referenzen zu arbeiten, birgt Gefahren, da Änderungen an einer Ebene zu fehlerhaften Referenzen führen können.

Eine separate Annotation für jede Informationsebene ist zwar relativ einfach zu realisieren, aber Primärdaten-Identität und Konsistenz zu erhalten liegt in der Verantwortung der Benutzer.

[1] siehe Abschnitt 2.1

### 2.3.2 Meilensteine

Meilensteine sind eine andere Art mit verschiedenen Informationsebenen und Überlappungen im Markup umzugehen. Sie sind leere Elemente, die Stellen im Dokument markieren, an denen eine Änderung auftritt.

```
1  <?xml version="1.0" encoding="iso-8859-1"?>
2  <lg>
3    <l><milestone type="start" gi="s" id="s1"/>
4        Sein Blick ist vom Vorübergehn der Stäbe
5    </l>
6    <l>so müd geworden, dass er nichts mehr hält.
7       </milestone type="end" gi="s" id="s1"/></l>
8    <l><milestone type="start" gi="s" is="s2"/>
9       Ihm ist, als ob es tausend Stäbe gäbe</l>
10   <l>und hinter tausend Stäben keine Welt.
11      </milestone type="end" gi="s" id="s1"/></l>
12 </lg>
```

Abbildung 2.5: Annotation der Versstruktur und der Satzebene unter Verwendung von Meilensteinen auf der Satzebene

Ein Beispiel für eine Annotation mit Meilensteinen zeigt Abbildung 2.5 unter Verwendung der Vorschläge aus den TEI Guidelines. Die primäre Annotationsebene bildet die Versstuktur. Das leere Element "`milestone`" markiert jeweils den Start- und Endpunkt der Region, über die sich das "`s`" Element spannen würde. Das "type" Attribut des "`milestone`" Elements gibt an, ob es sich um ein Start- oder ein Endtag handelt. Die ID-Attribute erlauben es die Paare von Meilenstein-Tags zu finden, die zusammengehören. Alternativ kann jedes Meilenstein-Tag eine eindeutige ID besitzen und die End-Tags verwenden ein zusätzliches Attribut "`coid`", das den Wert der ID des entsprechenden Start-Tags verweist.

Die Verwendung von Meilensteinen, auch wenn sie relativ einfach zu realisieren sind, hat einige Nachteile. Die Verarbeitung von Dokumenten mit Meileinsteinen ist problematisch, da sie aus XML-Sicht nur die Endpunkte einer Region markieren, die eigentlich als Element(-Inhalt) gesehen werden müsste. Die Verarbeitung mit Hilfe von XSL beispielsweise beruht jedoch auf den Inhalten und nicht auf den Endpunkten. Weiterhin sind Meilensteine recht unflexibel. Im Vorfeld der Annotation muss festgelegt werden, welche Informationsebene durch normale Elemente und welche durch Meilensteine realisiert werden soll. Ein Dokumentenschema muss entsprechend geschrieben werden, da ein nachträgliches Ändern nur schwer möglich ist. Außerdem sind die Anwender für die Analyse und Weiterverarbeitung der Dokumente auf eine spezielle Software mit dem Wissen über die Meilensteine und deren konkrete

Realisierung angewiesen, denn aus dem Dokument als solchem kann man diese Informationen nicht extrahieren.

Alternativ können Meilensteine auch als dedizierte Elemente realisiert werden. Es wäre auch möglich, "`s`" als Meilenstein-Element für die Start- und Endposition des Satzes zu verwenden. Natürlich könnte für Start und Ende auch dedizierte Elemente verwendet werden, zum Beispiel "`s-start`" und "`s-end`".

In [5] beschreibt DeRose eine weitere, Art Meilensteine zu realisieren. Anstatt eines generischen Meilenstein-Tags oder anderer spezieller Tags für Meilensteine soll genau das gleiche Tag für Meilensteine und normale Elemente eingesetzt werden. Soweit möglich soll die normale Element-Form (`<s>...</s>`) verwendet werden und wenn dies nicht möglich ist, sollen Meilensteine ( `<s sID='s1'/>...<s eID='s1'/>`) eingesetzt werden. DeRose nennt diesen Ansatz "Trojanische Meilensteine". Ein Vorteil dieser Methode ist, dass nur geringfügige Änderungen am Dokumentenschema notwendig sind und keine neuen Meilenstein-Tags eingeführt werden müssen. Die Benutzer können die bekannten Tags, abgesehen von einer kleinen Syntaxveränderung, (fast) wie gewohnt weiterverwenden.

Der Hauptnachteil anderer Meilensteinansätze bleiben jedoch bestehen. Normale XML Software kann die Dokument mit Meilensteinen nicht ohne weiteres korrekt verarbeiten. DeRose fasst vier Anforderungen für trojanische Meilensteine zusammen:

1. Elemente müssen leer sein, wenn ein "`sID`" oder "`eID`" Attribut vorhanden ist.

2. Wenn ein `eID` Attribute vorhanden ist, dürfen keine anderen Attribute vorhanden sein.

3. Jeder `sID`/`eID` Attribut-Wert darf nur zweimal im Dokument vorhanden sein und zwar genau einmal als `sID` Attribut und genau einmal als `eID` Attribut.

4. Leere Elemente mit übereistimmenden `sID` und `eID` Attributen dürfen nur in Paaren auftreten und in korrekter Reihenfolge.[2]

Diese Anforderungen können (komplett) von keiner aktuellen Schemasprache umgesetzt werden und müssen auf einer anderen Ebene durchgesetzt werden.

### 2.3.3 Fragmentierung

Eine weitere Möglichkeit, mit verschiedenen Informationsebenen in XML umzugehen, ist die Fragmentierung. An den Stellen, an denen es Überlappungen zwischen den Ebenen gibt, werden die Elemente einer Ebene segmentiert und in einzelne Teilstücke aufgeteilt.

[2] Das Element mit dem `sID` Attribut zuerst.

```
1  <?xml version="1.0" encoding="iso-8859-1"?>
2  <lg>
3    <l><s>Sein Blick ist vom Vorübergehn der Stäbe</s></l>
4    <l><s>so müd geworden, dass er nichts mehr hält.</s></l>
5    <l><s>Ihm ist, als ob es tausend Stäbe gäbe</s></l>
6    <l><s>und hinter tausend Stäben keine Welt.</s></l>
7  </lg>
```

Abbildung 2.6: Annotation der Versstruktur und der Satzebene unter Verwendung einer fragmentierten Satzebene

Die Abbildung 2.6 zeigt die Annotation mittels Fragmenten auf der Satzebene. Der Satz, der sich über die Zeilen 3 und 4 erstreckt, wird an der Stelle, wo sich die Annotation der Versstruktur mit dem Markup der linguististschen Struktur überschneidet, in zwei Teile bzw. Fragmente zerlegt. Das Start- und Endtag für das jeweilige Teilstück befindet sich jeweils auf der entsprechenden Zeile und ist von den "`l`" Tags der Versstruktur eingeschlossen. Gleiches gilt analog für die Zeilen 5 und 6.

Diese Methode ist einfach zu realisieren, aber auch sie bringt eine Reihe von Nachteilen mit sich. Zum einen kann die Semantik der Elemente, die fragmentiert werden, verloren gehen. Das "`s`" Element aus dem Beispiel beschreibt nicht mehr den ursprünglichen Satz, sondern nur noch ein Fragment. Auch muss wieder im Vorfeld festgelegt werden, welche Informationsebenen als Fragmente dargestellt werden sollen und welche nicht. Es gibt auch keine Information über den Zusammenhang der einzelnen Fragmente. Das Dokumentenschema muss entsprechend realisiert werden, und Änderungen sind relativ aufwendig.

### 2.3.4 Virtuelle Elemente

Virtuelle Elemente stellen eine Erweiterung der Fragmentierung dar. Die einzelnen Fragmente werden mit zusätzlichen Attributen versehen, die ein Zusammenfügen der einzelnen Fragmente ermöglichen.

Jedes "`s`" Element in Abbildung 2.7 hat ein zusätzliches Attribut "`id`" und optional die Attribute "`prev`" und "`next`". Mit dem "`prev`" Attribut verweist ein Fragment auf seinen Vorgänger und mit dem "`next`" auf seinen Nachfolger. Das erste und das letzte Fragment besitzen keine "`prev`" bzw. "`next`" Attribute.

Die TEI Guidelines schlagen alternativ das Verbinden der Fragmente durch Verwenden eines "`join`" Elements vor.

```
<join targets="s1_1 s1_2" result="s"/>
<join targets="s2_1 s2_2" result="s"/>
```

```
1  <?xml version="1.0" encoding="iso-8859-1"?>
2  <lg>
3    <l><s id="s1_1" next="s1_2">
4      Sein Blick ist vom Vorübergehn der Stäbe</s></l>
5    <l><s id="s1_2" prev="s1_1">
6      so müd geworden, dass er nichts mehr hält.</s></l>
7    <l><s id="s2_1" next="s2_2">
8      Ihm ist, als ob es tausend Stäbe gäbe</s></l>
9    <l><s id="s2_2" prev="s2_1">
10     und hinter tausend Stäben keine Welt.</s></l>
11 </lg>
```

Abbildung 2.7: Annotation der Versstruktur und der Satzebene unter Verwendung virtueller Elemente auf der Satzebene

Im Attribut "`targets`" werden alle IDs der Elemente durch Leerzeichen getrennt aufgeführt die verbunden werden sollen. Das Attribut "`result`" gibt den Namen des vebundenen Elements an.

Auch bei diesem Ansatz muss im Vorfeld festgelegt werden, welche Informationsebene durch Verwendung von normalen Elementen und welche durch virtuelle Elemente realisiert werden soll. Das Dokumentenschema muss entsprechend gestaltet werden. Weiterhin kommt man auch hier ohne eine Softwareunterstützung mit entsprechendem Wissen über die virtuellen Elemente bei der Verarbeitung nicht aus.

## 2.4 Layered Markup and Annotation Language

Einen nicht XML-basierten Ansatz stellt die Layered Markup and Annotation Language (LMNL) [19] dar. LMNL definiert primär ein Datenmodell und im Gegensatz zu XML[3] keine Syntax. LMNL bietet jedoch auch eine Syntax, die der von MECS [17] ähnlich ist.

Ein LMNL Dokument besteht aus verschiedenen Ebenen (Layers), die aufeinander aufbauen. Jede Ebene hat eine Basis (Base) und kann durch mehrere andere Ebenen überlagert sein (Overlay). Die unterste Basis ist die Text Ebene, die Text enthält. Jede Ebene, mit Ausnahme der Text Ebene, besteht aus verschiedenen Regionen (Ranges) die sich über die Inhalte ihrer untergeordneten Ebene erstrecken. Im Falle der Text-Ebene können die einzelnen Zeichen als Region gesehen werden. Eine Region hat einen Startpunkt und eine Länge. Beide Werte beziehen sich auf die Inhalte ihrer Basis. Bei einer Text-Ebene

[3]Erst das Document Object Model (DOM) und das XML Information Set geben XML ein Datenmodell.

sind es die einzelnen Zeichen (Characters). Bei einer anderen Ebene beziehen sich diese Werte auf die einzelnen Regionen ihrer Basis.

Regionen können benannt sein und Annotationen enthalten. Annotationen sind Attribut-Wert-Paare und ihr Wert entspricht einer Text-Ebene. Annotationen sind auf den ersten Blick mit Attributen aus der SGML/XML vergleichbar, aber sie können zusätzlich eine interne Struktur besitzen. Überlappungen sind in Annotatonen jedoch nicht zugelassen. Sie müssen hierarchisch aufgebaut sein.

```
1  [!lmnl version="0.2" encoding="iso-8859-1"]
2  [!layer name="sl" base="#text"]
3  [!layer name="ll" base="#text"]
4  [lg~sl}
5    [text~ll}
6      [l~sl}
7        [s~ll}Sein Blick ist vom Vorübergehn der Stäbe
8      {l~sl]
9      [l~sl}
10       so müd geworden, dass er nichts mehr hält.{s~ll]
11     {l~sl]
12     [l~sl}
13       [s~ll}Ihm ist, als ob es tausend Stäbe gäbe
14     {l~sl]
15     [l~sl}
16       und hinter tausend Stäben keine Welt.{s~ll]
17     {l~sl]
18   {text~ll]
19 {lg~sl]
20
```

Abbildung 2.8: Annotation des Gedichts mit LMNL

Abbildung 2.8 zeigt das Gedicht in LMNL annotiert. In den Zeilen 2 und 3 werden zwei Annotationsebenen deklariert, deren Basis jeweils die Text Ebene ist. Es werden, ähnlich wie in XML, Start- und End-Tags verwendet, die hier jedoch den Anfang und das Ende einer Region markieren. Jede Region wird explizit durch das Anhängen des Ebenennamens einer Ebene zugeordnet.

Neben Überlappungen auf verschiedenen Ebenen erlaubt LMNL auch überlappungen auf einer Ebene. Verschiedene Relationen zwischen einzelnen Elementen auf verschiedenen Ebenen sind in [7] beschrieben. Bei LMNL sind diese Relationen (Überlappungen, gleicher Start- oder Endpunkt, Identität, Überlappungen, ...) auch auf einer Ebene möglich.

### 2.4.1 Canonical LMNL In XML

In [5] beschreibt DeRose mit Canonical LMNL In XML (CLIX) eine XML-Syntax für LMNL. CLIX steht für "Canonical LMNL In XML". Mit Hilfe von trojanischen Meilensteinen kann ein LMNL Dokument so in ein wohlgeformtes XML-Dokument überführt werden. Der ursprüngliche Name HORSE (Hierarchy-Obfuscating Really Spiffy Encoding) wurde aufgebenen:

> With its spartan syntax it should be attractive; but its name seems less so, and cleanly a Muse meant us to rename this model CLIX because of its heavy use of point events scattered throughout the text or data stream: click, clicks, clix. (DeRose, [5])

Bei der Umwandlung von einem LMNL in ein CLIX Dokument werden die Primärdaten serialisiert und an den entsprechenden Stellen "CLIX", also Meilenstein-Elemente, eingefügt. Da LMNL in den Rangenamen bestimmte Zeichen erlaubt, die in XML nicht erlaubt sind, müssen diese anders repräsentiert werden. Dazu werden diese Zeichen in UTF-8 Codierung gewandelt und als String aus Hexadezimalzeichen[4], die durch zwei Bindestriche ("`-`") eingeschlossen sind, ausgegeben. Ein Bindestrich muss daher auch als ein entsprechender String ("`-2D-`") codiert werden.

Um eine eindeutige Überführung eines LMNL Dokument in ein CLIX Dokument zu erreichen, definiert DeRose sieben Schritte[5]. Um die resultierenden XML-Dokumente besser mit Standardsoftware verarbeiten zu können, sollten möglichst viele Elemente nicht als Meilensteine geschrieben werden. Diese Aufgabe ist jedoch nicht einfach zu lösen. DeRose gibt einen Algorithmus vor, der wohlgeformtes XML produziert, jedoch wird eine spezielle Ordnung, beispielsweise "x muss immer innerhalb von y liegen", nicht umgesetzt. Solche Regeln müssten, wenn möglich, aus dem Schema extrahiert werden und als erstes angewandt werden; dann könnte der Algorithmus das Resultat verarbeiten.

Abgesehen von LMNL würden sich auch SGML, XML, MECS und JITTs als CLIX Dokument darstellen lassen.

### 2.4.2 XML for Overlapping Structures

Ein weiterer Ansatz um LMNL in XML darzustellen stammt von Czmiel. In [4] beschreib er mit XML for Overlapping Structures (XfOS) eine entsprechende Syntax.

Ähnlich wie beim Ansatz von DeRose setzt Czmiel auf eine Technik, die auf Meilensteinen beruht. Die Start- und Endpunkte einer Region werden durch ein XML-Tag abgebildet. Der Name des Tags entspricht dem Namen

[4] Zahlen, die aus den Zeichen `0..9` und `A..F` aufgebaut sind.

[5] vgl. Abschnitt "Canonical ordering in CLIX" in [5]

der Range. Durch ein Attribut “type” wird jeweils angegeben, ob es sich bei dem Meilenstein-Element um den Start- oder Endpunkt einer Region handelt.

Wenn eine Region annotiert, also eine LMNL-Attribut (Annotation) am Start- oder Endpunkt enthält, wird diese als Inhalt des entsprechenden XML-Elements verwendet und nicht als XML-Attribut umgesetzt. Da LMNL-Annotationen hierarchisch seien müssen, können sie problemlos als Inhalt eines Elements verwendet werden. Die Annotationsnamen werden dabei zu XML-Elementen und die Werte wiederum zu Inhalten.

Das LMNL-Dokument mit zwei überlappenden Regionen

```
[a[x}1{]}o[b[y}2{]}ver{a]lap{b]
```

sieht in XfOS-Syntax wie folgt aus:[6]

```
<a type="start"><x>1</x></a>o<b type="start"><y>2</y>
  </b>ver<a type="end"/>lap<b type="end"/>
```

Durch XfOS ergibt sich eine weitere Möglichkeit einer XML-Syntax für LMNL. Jedoch kann XfOS nicht alle Eigenschaften von LMNL modellieren. Es ist beispielweise nicht möglich, unbenannte Regionen zu erstellen oder Entitäten irgendwo im Dokument zu deklarieren.

[6] Der Zeilenumbruch im Start-Meilenstein der “b”-Region erfolgt nur, weil die das Beispiel nicht auf eine Zeile passen würde.

# Kapitel 3

# Die XML-CONCUR-Dokument-Syntax

Die XML basierten Ansätze zur multi-hierarchischen Annotation aus Abschnitt 2.3 haben meist das Problem, daß Benutzer sich bereits im Vorfeld der Annotation für eine primäre Ebene entscheiden müssen. Eine späteres Wechseln der Primärsicht auf die Daten ist nur schwer realisierbar und die Verwendbarkeit der Annotation wird so eingeschränkt. LMNL aus Abschnitt 2.4 priorisiert keine Ebene, aber der syntaktische Unterschied zu XML ist sehr gross. Die SGML-CONCUR-Syntax bietet eine Lösung für beide Probleme. Zum einen sind alle Annotationsebenen gleichberechtigt notiert und zum anderen hat SGML-CONCUR eine große Ähnlichkeit mit XML. Die Verwendung von Dokumenttyp-Spezifikationen unterscheidet sich in der Anwendung nicht maßgeblich von der Verwendung von XML-Namespaces. Für Anwender, die zwar keine Erfahrung mit SGML haben, jedoch über XML Kentnisse verfügen, sollte dies nicht zu nennenswerten Problemen führen.

Die XML-CONCUR-Dokument-Syntax ist eine an SGML-CONCUR angelehnte Syntax für multi-hierarchisch strukturierte Daten. Die Grundlagen dazu werden in [11, 12] gelegt. Die Syntax sollte möglichst nahe an XML angelehnt sein, um den Anwendern einen leichten Einstieg zu ermöglichen. Der ursprüngliche Name MuLaX (Multi-Layered-XML) wurde zugunsten von XML-CONCUR (XMC) aufgegeben.

Ein XMC-Dokument kann als eine Vermischung eines Satzes primärdatenidentischer XML-Dokumente gesehen werden. Da XML gegenüber SGML gewissen Einschränkungen unterliegt, unterliegen auch die einzelnen Annotationsebenen, und damit das gesamte XMC-Dokument, diesen Einschränkungen.

Für die Beschreibung der XMC-Dokument-Syntax wird die Terminologie der XML Recommendation ([2]) verwendet. Im weiteren werden noch einige andere Begriffe definiert.

## 3.1 Elemente der XMC-Syntax

Der technische Begriff der *Annotationsebene* bezeichnet die konkrete syntaktische Umsetzung einer Informationsebene durch Annotationen in einem Dokument. Eine Annotationsebene kann auch mehrere Informationsebenen realisieren. Wenn man ein XMC-Dokument als Vermischung einzelner, primärdatenidentischer XML-Dokumente sieht, entspricht jede Annotationsebene einem XML-Dokument.

Ein *Annotationschema* beschreibt die Elemente und ihre Relationen zueinander, die in einer Annotationsebene verwendet werden können. In einem XMC-Dokument kann ein Annotationschema *implizit* verwendet oder *explizit* deklariert werden. Ein implizites Schema wird angenommen, wenn Elemente und Attribute ohne vorherige Definition verwendet werden. Wie bei XML kann so eine Annotation erfolgen, ohne vorher ein Schema definieren zu müssen.

Bei der Verwendung eines expliziten Schemas wird dies am Beginn eines XMC-Dokuments deklariert. Das Benutzen eines expliziten Schemas ermöglicht es, die Annotation zu validieren, also auf die richtige Verwendung von Elementen und Attributen zu prüfen. Der Gebrauch verschiedener Schemasprachen ist nicht direkt durch die XMC Dokument-Syntax reglementiert, daher können alle Schemasprachen wie DTDs, XML-Schema und RelaxNG verwendet werden.

### 3.1.1 Annotationsschema-Deklaration

Um ein explizites Annotationsschema zu verwenden, muss es deklariert werden. Durch eine *Annotationsschema-Deklaration* wird für eine Annotationsebene ein Annotationsschema festgelegt. Eine XMC Annotationschema-Deklaration sieht wie folgt aus:

```
<!DOCTYPE (2)text SYSTEM "teiana2.dtd">
```

In diesem Beispiel wird einer Annotationsebene eine DTD, die in der lokalen Datei teiana2.dtd vorliegt, zugewiesen. Gleichzeitig wird auch die Annotationsebenen-Id “2” für diese Ebene festgelegt, die in runden Klammern vor dem Wurzelelement `text` der Ebene angegeben wird. Die Wahl der Schemasprache ist nicht auf DTD oder DTD Fragemente beschänkt; XML-Schema und RelaxNG können auch verwendet werden. Interne DTDs sind ebenfalls erlaubt. Neben dem `SYSTEM` kann auch ein `PUBLIC` Identifier verwendet werden.

Die Public- und System-Identifier erlauben es, ein Annotationschema zu referenzieren. Ein System-Identifier ist ein Uniform Resource Identifier (URI) und kann auf eine beliebige Datei im lokalen Dateisystem des Computers oder auf eine Internet-Ressource verweisen.

Für standardisierte DTDs gibt es oft auch einen Public-Identifier. Dies ist eine eindeutige Zeichenkette[1], die einen festgelegten Dokumenttyp beschreibt. Ein Programm kann entweder die zu einem Public-Identifier gehörenden DTD fest eingebaut haben oder einen Katalog nutzen, der den Public-Identifier auf ein entsprechende Datei abbildet.

### 3.1.2 Elemente

Vor jedem Element wird, ähnlich wie bei SGML-CONCUR, in runden Klammern eine *Annotationsebenen-Id*, auch *Annotationsebenen-Prefix* genannt, angegeben. Durch eine Annotationschema-Deklaration wird eine Annotationsebenen-Id einem Schema und damit einer Annotationsebene zugeordnet. Die Elemente und ihre Relationen zueinander können dann anhand des Schemas auf Korrektheit geprüft werden. Für eine Annotationsebenen-Id gelten die gleichen Konventionen, die auch für Attributnamen in XML gelten.

Jedes XMC-Dokument muss "wohlgeformt" sein. Das bedeutet, dass die *Projektion* eines wohlgeformten XMC-Dokuments auf eine Annotationsebene ein wohlgeformtes XML-Dokument erzeugen muss. Eine Projektion auf eine Annotationsebene ist von Hilbert in [11] wie folgt definiert:

1. Entferne alle Elemente, die nicht zu der Annotationsebene gehören, auf die projiziert werden soll, d. h. alle Elemente, deren Annotationsebenen-Id nicht mit der Annotationsebenen-Id der zu Projizierenden übereinstimmt. Entferne alle Annotationsschema-Deklarationen, die nicht zu der Annotationsebene gehören, auf die projiziert wird.

2. Entferne alle Annotationsebenen-Ids von den Elementen und einer eventuell vorhandenen Annotationsschema-Deklaration. Ersetze die XMC-Deklaration durch eine XML-Deklaration.

Neben der obligatorischen Eigenschaft der Wohlgeformtheit definiert XMC eine optionale "Validität". Da für eine Annotationsebene kein Annotationsschema vorliegen muss, definiert Hilbert ([11]) für XMC zwei Validitätsbegriffe: die Validität auf einer Annotationsebene und die allgemeine Validität.

**Definition 1** *Ein XMC-Dokument heisst* valide auf einer Annotationsebene *genau dann, wenn für diese Ebene ein Annotationsschema vorliegt und die Projektion des Dokuments auf diese Ebene ein valides XML-Dokument ergibt.*

**Definition 2** *Ein XMC-Dokument heisst* allgemein valide *genau dann, wenn es auf allen Annotationsebenen valide ist.*

[1] Zum Beispiel "`-//W3C//DTD XHTML 1.0 Strict//EN`" für XHTML.

Die XMC-Dokumenten-Syntax ist, verglichen mit SGML-CONCUR, in einigen Aspekten strikter und in anderen weniger strikt. Im Bezug auf die Schemadeklaration ist XMC weniger strikt, da eine explizite Deklaration in XMC keine Pflicht ist. Anderseits erlaubt XMC keine gemeinsamen Elemente für Annotationsebenen. Wenn auf verschiedenen Annotationsebenen das gleiche Element vorkommt, muss es in XMC explizit angeben werden. Jedem Element muss eine Annotationsebenen-Id voranstehen und jede Annotationsebene muss ein Wurzelelement haben.

Diese Einschränkungen erleichtern die Entwicklung und Implementierung eines Verarbeitungsmodells. Besonders die Verwendung von gemeinsamen Elementen würde ein solches Verarbeitungsmodell komplizierter machen. Gemeinsame Elemente könnten durch mehrere Objekte, je eins für eine Annotationsebene, im Verarbeitungsmodell repräsentiert werden oder es wird ein Objekt verwendet, das von allen Ebenen verwendet wird. Beide Lösungsmöglichkeiten erhöhen jedoch die Komplexität des Verarbeitungsmodells, da entweder die verschiedenen Objekte mit den Ebenen synchronisiert werden müssen oder das Objekt von mehreren Ebenen referenziert wird.

### 3.1.3 Attribute

Im Gegensatz zu Elementen haben Attribute keine Annotationsebenen-Id, da sie immer einem Element zugeordnet werden. Wenn auf zwei Annotationsebenen an derselben Stelle[2] das gleiche Element mit jeweils einem unterschiedlichen Attribut vorkommt, können diese nicht zu einem Element mit zwei Attributen zusammengefasst werden, weil XMC keine gemeinsamen Elemente zulässt. Daher werden die Elemente

```
<p lang="de">...</p>
```

und

```
<p align="center">...</p>
```

zu folgendem zusamengeführt

```
<(1)p lang="de"><(2)p align="center">...<(2)/p><(1)/p>
```

und *nicht* zu

```
<p (1)lang="de" (2)align="center">...</p>
```

[2] Auf die Primärdaten bezogen.

### 3.1.4 Kommentare

In einem XMC-Dokument können, ebenso wie in XML, *Kommentare* eingebettet sein. Ein Kommentar kann entweder nur *lokal* für eine Annotationsebene oder *global* für das gesamte Dokument gelten. Wenn er einer Annotationsebene zugeordnet ist, muss ihm eine Annotationsebenen-Id vorangestellt werden; wenn er global ist, ist dies nicht notwendig. Ein globaler Kommentar ist jedoch nicht mit einem Kommentar auf allen Annotationsebenen gleichzusetzen; er gilt für das XMC-Dokument und ist deswegen keiner Annotationsebene zugewiesen.

Folgende Syntax ist für einen Kommentar auf der Annotationsebene `aid` zu verwenden:

```
<(aid)!-- ein lokaler Kommentar -->
```

Ein globaler Kommentar unterscheidet sich syntaktisch nicht von einem normalen Kommentar in XML:

```
<!-- ein globaler Kommentar -->
```

### 3.1.5 Verarbeitungsanweisungen

Analog zu Kommentaren unterstützt XMC auch Verarbeitungsanweisungen (Processing Instructions), die sowohl *lokal* als auch *global* definiert sein können. Einer lokalen Verarbeitungsanweisung wird, ebenso wie beim Kommentar, eine Annotationsebenen-Id vorangestellt. Ebenso wie ein Kommentar wird eine globale Verarbeitungsanweisung keiner Annotationsebene zugewiesen und ist daher auch nicht einer Verarbeitungsanweisung auf allen Annotationsebenen gleichzusetzen.

Die Syntax für eine lokale Verarbeitungsanweisung auf Ebene `aid` sieht wie folgt aus:

```
<(aid)?php echo "Hallo XMC-Welt!"; ?>
```

Für eine globale Verarbeitungsanweisung ist die bekannte XML-Syntax zu verwenden:

```
<?php echo "Hallo XMC-Welt!"; ?>
```

In [12] wird eine leicht abgewandelte Syntax für Kommentare und Verarbeitungsanweisungen vorgeschlagen (Annotationsebenen-Id bei Kommentaren nach den Zeichen "`!-`" und bei Verarbeitungsanweisungen nach "`?`".). Der Vorschlag in dieser Arbeit vereinfacht die Anwendung der Kommentare und Verarbeitungsanweisungen, da die Annotationsebenen-Id wie bei den Tags direkt nach der öffnenden spitzen Klammer angegeben wird. Die Dokument-Syntax ist so konsistenter.

Damit Kommentare und Verarbeitungsanweisungen richtig im Verarbeitungsmodell definiert sind, muss der Projektion eines XMC-Dokuments auf eine Annotationsebene folgendes hinzugefügt werden: Entferne alle globalen Kommentare und Verarbeitungsanweisungen. Entferne alle Kommentare und Verarbeitungsanweisungen, die nicht zu der Annotationsebene gehören, auf die projiziert wird. Entferne alle Annotationsebenen-Ids von den verbliebenen Kommentaren und Verarbeitungsanweisungen.

### 3.1.6 Entitäten

*Entitäten* sind für XMC *nicht* definiert. Dies hat zwei Gründe. Zum ersten ist es nur in DTDs möglich, Entitäten zu deklarieren. Wenn beispielsweise eine Entität in einem Dokument mit einer XML-Schema Grammatik verwendet werden sollte, müsste sie entweder in einem internal (DTD) Subset definiert werden oder im Schema ein Element definieren, dessen "`fixed`" Inhaltsmodell dem Wert der Entität entspricht. Letzteres muss jedoch dann auch wie ein leeres Element und nicht wie eine normale Entität verwendet werden. In [8] Anhang C "Using Entities" sind beide Vorgehensweisen beschrieben.

Zweitens sind alle Entitäten in XMC auf allen Annotationsebenen definiert, da sie in den Primärdaten verwendet werden, und diese sind definitionsbedingt für alle Ebenen gleich. So müsste eine Entität in allen verwendeten Annotationschemata deklariert sein. Das ist einerseits vielleicht nicht gewollt, da eventuell mehrere Schemata modifiziert werden müssten und anderseits ist es, wie oben bereits angedeutet, nicht in allen Schemasprachen ohne weiteres möglich, Entitäten zu definieren.

## 3.2 Ein Beispiel für ein XMC-Dokument

In Abbildung 3.1 ist die erste Strophe von Rilkes Gedicht "Der Panther" in XMC annotiert. Es ist eine Annotation auf zwei Annotationsebenen vorgenommen worden: zum einen die lyrische und zum anderen die linguistische Struktur.

In Zeile 1 befindet sich die obligatrische XMC-Deklaration. Sie legt die XMC-Version und die verwendete Kodierung fest. Die Angabe der Version ist zwingend erforderlich und muss den fest definierten Wert "1.0" haben. Wenn keine Kodierung angeben ist, wird UTF-8 angenommen. Für die Werte, die bei der Angabe der Kodierung verwendet werden können, gelten die gleichen Konventionen wie bei XML.

In den Zeilen 2 und 3 sind zwei Annotationsschema-Deklarationen angegeben: die Annotationsebene mit der Annotationsebenen-Id "1" und dem Wurzelelement "`lg`", die die DTD aus der Datei teivers2.dtd verwendet, und die Annotationsebene mit der Annotationsebenen-Id "2" und dem Wurzelelement "text", die die DTD aus der Datei teiana2.dtd verwendet. Die entsprechenden

```
1  <?xmc version="1.0" encoding="iso-8859-1"?>
2  <!DOCTYPE (1)lg SYSTEM "teivers2.dtd">
3  <!DOCTYPE (2)text SYSTEM "teiana2.dtd">
4  <(1)lg>
5    <(2)text>
6      <(1)l>
7        <(2)s>Sein Blick ist vom Vorübergehn der Stäbe
8      </(1)l>
9      <(1)l>
10       so müd geworden, dass er nichts mehr hält.</(2)s>
11     </(1)l>
12     <(1)l>
13       <(2)s>Ihm ist, als ob es tausend Stäbe gäbe
14     </(1)l>
15     <(1)l>
16       und hinter tausend Stäben keine Welt.</(2)s>
17     </(1)l>
18   </(2)text>
19 </(1)lg>
```

Abbildung 3.1: “Der Panther” in XMC annotiert

Annotationsebenen-Ids werden in allen Elementen in runden Klammern vor dem Elementnamen angegeben.

In Zeile 4 wird das Wurzelelement für die Annotationsebene der Versstruktur, die die Annotationsebenen-Id “1” besitzt, geöffnet und in Zeile 19 wieder geschlossen. Das Wurzelelement für die Annotationsebene der linguistischen Struktur mit der Annotationsebenen-Id “2” wird auf Zeile 5 geöffnet und in Zeile 18 geschlossen.

Die einzelnen Verse des Gedichts werden jeweils mit “`l`” Elementen und die beiden Sätze der Gedichts durch jeweils ein “`s`” Element annotiert.

In den Zeilen 7 bis 10 gibt es eine Überlappung in der Annotation. Der erste Satz des Gedichts erstreckt sich über die ersten beiden Zeilen. Die Annotation für die erste Zeile wird mit einem End-Tag in Zeile 9 geschlossen; in Zeile 10 steht das Start-Tag für die Annotation der zweiten Zeile des Gedichts. Der Satz, und damit auch dessen Annotation, erstreckt sich über diesen Zeilenwechsel und die damit verbundene Annotation hinweg.

## 3.3 XML-CONCUR vs. XML-Namespaces

Auf den ersten Blick scheint das Konzept der Annotationsebenen in XMC und XML-Namespaces sehr ähnlich zu sein, da beide es ermöglichen, Elemente

aus verschiedenen Annotationsschemata in einem Dokument zu verwenden. Jedoch auch mit XML-Namespaces muss das XML-Dokument hierarchisch strukturiert sein und Überlappungen sind nicht möglich.

Die Kennzeichnung der Annotationsebenen durch eine vorangestellte Annotationsebenen-Id in runden Klammern im Gegensatz zu einem erweiterten Namespace-Ansatz hat noch einen weiteren Vorteil. In einem XMC-Dokument können auf einer Annotationsebene Elemente aus verschiedenen Namespaces verwendet werden. Ein Benutzer kann durchaus Gründe haben, ein Dokument so gestalten zu wollen, und es wäre eine unnötige Einschränkung, wenn XMC dies nicht erlauben würde.

Weiterhin kann so auch das gleiche Annotationsschema auf verschiedenen Annotationsebenen verwendet werden. Ein Namespace ist immer zu einem definierten Namespace-Namen zugeordnet, der, zumindest konzeptuell, auf ein Schema oder Tagset verweist, in dem die Elemente definiert sind. Durch die Annotationsebenen-Id kann in XMC ein Schema für verschiedene Ebenen verwendet werden; dies ermöglicht es eine alternative Annotation mit den gleichen Elementen oder verschiedene Subsets eines Schemas vorzunehmen.

# Kapitel 4

# XML-CONCUR-Verarbeitungsmodelle

Der nachfolgende Abschnitt stellt zwei verschiedene Verarbeitungsmodelle für XMC-Dokumente vor. Der erste Ansatz stammt von Mirco Hilbert und bescheibt eine hierarchische Struktur mit Meilenstein-Elementen[1]. Der zweite Ansatz wurde im Rahmen dieser Arbeit entwickelt und definiert ein lineares Modell mit einer hierarchischen Struktur für die einzelnen Annotationsebenen. Die Modelle sind primär für einen Editor, das heisst eine Anwendung zum Bearbeiten von XMC-Dokumenten, konzipiert.

## 4.1 Just-In-Time-Trees (JITTs)

In [11, 12] beschreibt Hilbert ein Verarbeitungsmodell für XMC. Es definiert eine hierarchische Struktur von Objekten, die dem Document Object Model (DOM, [13]) ähnelt.

Dieses Modell entstand vor der Umbenennung von MuLaX in XML-CONCUR, daher ist "MuLaX" ein Bestandteil der Namen von vielen Objekten in der Implementierung.

### 4.1.1 Konzeptuell

Das Verarbeitungsmodell stellt eine hierarchische Sicht auf das Dokument aus dem Blickwinkel einer Annotationsebene dar. Um das Dokument aus der Sichtweise aller Annotationsebenen darzustellen, würden $n$ Instanzen des Modells gebraucht, wenn das Dokument $n$ Annotationsebenen besitzt. Jedoch, aus der Sicht eines Editors, arbeitet der Benutzer zu einem Zeitpunkt nur auf einer Annotationsebene, daher wird nur eine Instanz gebraucht. So entfällt der Auf-

[1] siehe Abschnitt 2.3.2

wand und die Komplexität einer Synchronisierung zwischen den einzelnen Instanzen.

Da jeweils das Dokument nur aus der Sicht einer Ebene gesehen wird, unterscheidet Hilbert zwischen der *aktuellen Annotationsebene* ("current annotation layer") und den *fremden Annotationsebenen* ("foreign annotation layers")

Die aktuelle Annotationsebene ist die Ebene aus deren Sicht das Dokument zu einem Zeitpunkt betrachtet wird. Sie wird durch eine hierarchische Knotenstruktur gebildet. Die fremden Annotationsebenen sind die anderen Ebenen. Sie werden als Meilenstein-Elemente betrachtet und durch entsprechende Elemente in die hierarchische Struktur der aktuellen Annotationsebene eingebunden.

Traditionell musste man sich beim Erstellen eines Annotationsschemas, beispielsweise einer DTD, überlegen, welche Elemente als normale Elemente verwendet werden sollen, und welche als Meilensteine. Mit dem hier beschriebenen Verarbeitungsmodell muss diese Entscheidung nicht getroffen werden, da jede Annotationsebene (zu unterschiedlichen Zeiten) in der Rolle als aktuelle oder als fremde Annotationsebene auftreten kann. Das Erzeugen von virtuellen Elementen durch Verlinkung mit IDREF-Mechanismen und die Schwierigkeiten, diese konsistent zu halten entfallen, mit MuLaX.

Einige Aspekte dieses Modells entsprechen der Idee der Just-In-Time-Trees die Durusau in [6, 7] thematisiert. Durusau argumentiert, dass ein bestimmter Wurzelknoten und Markup eigentlich erst benötigt werden, wenn ein Dokument verarbeitet werden soll. Verschiedene Zielsetzungen bei der Verarbeitung des Dokuments können eine unterschiedliche Annotation erfordern. Mit Just-In-Time-Trees wird die relevante Annotation erst bei der Verarbeitung deklariert bzw. ausgewählt und so kann beispielsweise dem Problem der Überlappung aus dem Weg gegangen werden.

Das Verarbeitungsmodell von Hilbert ermöglicht es, eine bestimmte Ebene als primäre Annotationsebene auszuwählen. So kann jeweils die Instanz des Verarbeitungsmodells gewählt werden, die für eine bestimmte Zielsetzung am Besten geeignet ist. Die Entscheidung, welche Instanz am geeignetesten ist, wird, analog zu den Just-In-Time-Trees von Durusau, aufgeschoben und erst bei der Verarbeitung gestellt.

### 4.1.2 Struktur

Das Verarbeitungsmodell definiert, ähnlich wie das DOM[13], eine hierarchische Struktur von Knoten mit verschiedenen Typen. Es sind Knoten-Typen für das Wurzelelement, die normalen Elemente, Attribute, Text-Elemente und die Knoten einer fremden Annotationsebene ("foreign-nodes") definiert. Es sind zur Zeit keine Knoten für Kommentare oder Verarbeitungsanweisungen (Processing-Instructions) definiert. Jeder Knoten, mit Ausnahme des Wurzelknotens, hat genau einen Elternknoten. Bis auf die Textknoten gehört jeder

Knoten zu einer Annotationsebene.

Der *MuLaXModel*-Knoten ist der Wurzelknoten. Er repräsentiert das gesamte Dokument aus der Sicht der aktuellen Annotationsebene. MuLaX verlangt ein Wurzelelement pro Annotationsebene. Ein MuLaXModel-Knoten kann daher nur ein MuLaXElement-Knoten als Kind besitzen. Dieser kann von MuLaXForeignStart-, MuLaxForeignEnd- oder MuLaXForeignEmpty-Knoten umgeben sein. Es kann kein MuLaXText-Knoten als Kind vorhanden sein, da dieser Text ausserhalb des Wurzelknotens für die aktuelle Annotationsebene läge. Der MuLaXModel-Knoten speichert die Annotationsebenen-Id der aktuellen Annotationsebene.

Die Elemente werden durch den *MuLaXElement*-Knoten abgebildet. Er besitzt eine Liste von Kindknoten, die der Reihenfolge, in der sie im Dokument auftreten. Wenn der Knoten keine Kinder hat, also ein Empty-Tag ist, ist die Liste leer. Als Kindknoten sind MuLaXText-, MuLaXForeignStart-, MuLaXForeignEnd-, MuLaXForeignEmpty- und weitere MuLaXElement-Knoten erlaubt. Zusätzlich kann ein MuLaXElement-Knoten eine Liste von MuLaXAttribut-Knoten haben. Wenn das Element keine Attribute hat, ist die Liste leer. Jeder MuLaXElement-Knoten gehört zu einer Annotationsebene und besitzt eine Annotationsebenen-Id. Weiterhin hat jedes MuLaXElement einen Namen ("generic identifier").

Ein *MuLaXAttribute*-Knoten bildet Attribute ab. Er speichert einen Namen und einen Wert, für die die gleichen Restriktionen gelten wie bei XML. MuLaXAttribute-Knoten haben immer einen MuLaXElement-Knoten als Elternknoten.

Der *MuLaXText*-Knoten gehört allen Annotationsebenen an. Ihm wird eine künstliche Annotationsebenen-Id von 0 zugewiesen. Der Elternknoten ist immer der übergeordnete MuLaXElement-Knoten. Wenn eine andere Annotationsebene als aktuelle Ebene gewählt wird, muss eventuell ein anderer Elternknoten gewählt werden. Der Wert des Knotens entspricht den Primärdaten. Analog zum DOM erlaubt das Verarbeitungsmodell keine direkt nebeneinander liegenden MuLaXText-Knoten. Es muss immer mindestens ein MuLaXElement oder einer der "foreign"-Knoten zwischen ihnen stehen.

Die "foreign"-Knoten-Familie definiert die Meilenstein-Elemente der Annotationsebenen, die nicht die "aktuelle" sind. Entsprechend der Tag-Typen, die es in XML gibt, sind drei verschiedene Knoten definiert: *MuLaXForeignStartTag*, *MuLaXForeignEndTag* und *MuLaXForeignEmptyTag*. Sie haben keine Kindknoten und referenzieren immer eine "foreign" Annotationsebene. Analog zum MuLaXElement-Knoten können der MuLaXForeignStartTag- und der MuLaXForeignEmptyTag-Knoten einen oder mehrereAttribut-Knoten haben.

Die Zusammengehörigkeit von einem MuLaXForeignStartTag- und einem MuLaXForeignEndTag-Knoten kann durch Verweise implementiert werden, aber diese Eigenschaft ist von einem MuLaX-Parser nicht zwingend gefordert. Da jedoch ohne solche Verweise nicht geprüft werden kann, ob zu jedem

Start-Tag auch ein End-Tag vorhanden ist, kann nicht auf allen Annotationsebenen eine Prüfung auf Wohlgeformtheit vorgenommen werden. Der höhere Aufwand, der für das Finden der Paarbeziehungen der Start- und End-Tags getrieben werden muss, relativiert sich, wenn man dafür auf allen Ebenen auf Wohlgeformtheit prüfen kann.

## 4.2 Real-Time-Trees (RTT)

Dieses Verarbeitungsmodell wurde im Rahmen dieser Arbeit entwickelt. Im Gegensatz zum Just-In-Time-Tree-Ansatz wird hier ein Modell verwendet, dass auf mehreren, unabhängigen hierarchischen Strukturen aufbaut.

### 4.2.1 Konzeptuell

Es werden zwei unterschiedliche Datenbegriffe in diesem Verarbeitungsmodell unterschieden: *Primärdaten* und *Dokumentdaten*. Ebenso wie in Abschnitt 2.1 umfasst der Begriff Primärdaten das Urspungsdokument, also das Dokument, das entsteht, wenn das gesamte Markup entfernt wird. Der Begriff Dokumentdaten hingegen umfasst das Urspungsdokument inklusive dem Markup.

Das Verarbeitungsmodell besteht aus zwei Teilen und definiert dafür unterschiedliche Strukturen: die *lineare Struktur* und die *hierarchische Struktur*.

Für die lineare Struktur werden die Dokumentdaten in verschiedene Abschnitte geteilt, die *Segmente* genannt werden. Ein Abschnitt kann beispielsweise einem Start-Tag oder einem Text-Abschnitt entsprechen. Die einzelnen Segmente werden aus einem Abschnitt von einzelnen Zeichen des Dokuments gebildet. Je nachdem welchen Inhalt sie repräsentieren, wird ihnen eine Annotationsebene zugewiesen oder sie sind auf allen Ebenen verfügbar.

Neben der linearen Struktur gibt es für jede Annotationsebene eine hierarchische Struktur. Sie ist, analog zum DOM und dem vorherigen Just-In-Time-Tree-Ansatz, aus Knoten aufgebaut. So existieren zu jeder Instanz des Verarbeitungsmodells alle hierarchischen Strukturen für alle Annotationsebenen gleichzeitig und gleichberechtigt nebeneinander. Aus dieser Eigenschaft leitet sich auch der Name des Modells ab.

Ein ähnliches Konzept der Segmentierung der Dokumente in einzelne Teilstücke wird auch bei LMNL (siehe Abschnitt 2.4, [19]) verwendet. Mit der Core Range Algebra von Nicol ([14]) wird dafür eine formale Grundlage und ein Formalismus für Operationen auf Teilstücken (Ranges) gelegt Es werden die grundlegenden Datentypen *Sequence* und *Range* definiert.

Eine Sequenz ist eine endliche Liste von Elementen, die einer Ordnung unterliegen. Es sind verschiedene Preädikate (IsEmpty, Contains, Equals), Operatoren (Diff, Union, Intersect, Concat) und Operationen (Insert, Delete, ItemAt, Last, First) für Sequenzen definiert.

Eine Range ist als Sub-Sequenz definiert und beschreibt eine Start-Position und eine Länge innerhalb einer Sequenz. Auch hier sind eine Reihe von Prädikaten (StartsBefore, StartsAfter, StartsWithin, EndBefore, EndsAfter, EndsWithin, Within, Same) und Operationen (Create, Flatten, EndOf, Move, Resize) definiert.

Neben den genannten Datentypen gibt es noch Sequences of Ranges. Die Funktionen (StartOrder, EndOrder, Unqiue, Ancestor, Descendant, Parent, Child) werden verwendet, um die Relationen der Ranges untereinander zu beschreiben. Durch Beschreibung, wie einzelne Ranges andere einschliessen, kann eine Struktur des Dokuments inferiert werden.

Die lineare Struktur des hier beschriebenen Verarbeitungsmodells verwendet mit den Segmenten ein ähnliches Konzept wie LMNL mit der Core Range Algebra und den dort definierten Sequenzen. Im Gegensatz zu LMNL werden hier jedoch nicht nur Ranges über die Primärdaten gebildet, sondern auch über das Markup. Es sind jedoch nur Ranges über Zeichen erlaubt und keine Sequenzen über andere Sequenzen. Somit entspricht das Verarbeitungsmodell in LMNL am ehesten einer Sequenz von Ranges über die Text-Ebene.

### 4.2.2 Struktur

Die Real-Time-Trees-Verarbeitungsmodell definiert zwei Strukturen: eine Sequenz von Segmenten und eine, an das DOM angelehnte, Hierarchie von Knoten mit verschiedenem Typ für jede Annotationsebene.

Das *MultiLayerDocument* Objekt bildet die Basis für das Modell. Es enthält die Objekte der lineare und die hierarchischen Strukturen und Informationen über die Annotationsebenen.

Die lineare Struktur wird durch eine einfache Liste von *DocumentItem* Objekten gebildet. Jedes dieser Objekte bildet genau ein Teilstück, auch *Segment* genannt, über den Dokumentdaten[2] ab. Sie besitzen einen *Offset*, der den Startpunkt des Segments bestimmt, und eine *Länge*, die die Länge des Segments angibt. Jedem Segment wird ein Typ zugewiesen, der sich danach richtet, was die Zeichenkette, die das Segment bildet, repräsentiert. Der Typ kann *ElementStart* für ein Start-Tag, *ElementEnd* für ein End-Tag, *ElementEmpty* für leeres Element und *Text* für ein Segment, das nur Text enthält, sein.

Zusätzlich gibt es noch einen Typ *Ignore*, der zur Zeit für Segmente verwendet wird, die einer XMC-Deklaration oder einer Annotationsschema-Deklaration entsprechen. In einer zuküftigen Version des Verarbeitungsmodells werden dafür eigene Typen definiert.

Einem Segment vom Typ ElementStart, ElementEnd und ElementEmpty wird eine Annotationsebenen-Id zugeordnet, um sie einer Annotationsebene

[2] siehe Abschnitt 4.2.1

zuzuordnen. Den Segmenten von Typ Text und Ignore kann keine Annotationsebenen-Id zugewiesen werden, da sie, im Falle vom Typ Text, auf allen Annotationsebenen existieren oder, im anderen Fall, vom Modell ignoriert werden sollen. Ihnen wird daher der spezielle Wert *every* als Annotationebenen-Id zugewiesen, der bedeutet, dass diese Segmente auf allen Ebenen vorhanden sind[3]. Diesen Wert, der nur intern verwendet wird, kann keine reguläre Annotationsebenen-Id annehmen. Er ist nur intern für das Verarbeitungsmodell definiert. Es ist durchaus möglich, eine Annotationsebene mit einer Annotationsebenen-Id "`every`" zu haben.

Die Attribute der Start-Tags werden von *Attribute* Objekten abgebildet. Jedes DokumentItem vom Typ ElementStart kann eine Liste von Attributen verwalten. Ein Attribut hat einen Namen und einen Wert und für beide gelten die gleichen Restriktionen wie bei XML.

Die hierarchische Struktur wird durch eine Baumstruktur mit Knoten von verschiedenem Typ gebildet. Es gibt für jede Annotationsebene eine Struktur.

Eine Annotationsebene wird durch das *AnnotationLayer* Objekt abgebildet. Für jede Annotationsebene existiert eines dieser Objekte innerhalb des MultiLayerDocuments. Gleichzeitig stellt dieses Objekt das Äquivalent zum Document-Knoten des DOM dar und bildet den übergeordneten Knoten für die Annotationsebenen. Er besitzt eine Annotationsebenen-Id und hat genau einen Kind-Knoten, der das Wurzelelement der entsprechenden Annotationsebene bildet.

Der *ElementNode*-Knoten bildet die Elemente ab. Er verweist auf die beiden DocumentItem Objekte, deren Segmente das entsprechende Start- und End-Tag bilden. Ein ElementNode-Knoten hat eine Liste von Kindknoten, die in der Reihenfolge sortiert sind, wie sie im Dokument stehen. Wenn es keine Kind-Knoten gibt, ist die Liste leer. Weiterhin besitzt der Knoten noch einen Verweis auf den Eltern-Knoten. Wenn ein Element-Knoten der Wurzelknoten einer Annotationsebene ist, gibt es keine übergeordneten Knoten und der Verweis ist leer.

Ein leeres Element wird mit dem *EmptyElementNode*-Knoten abgebildet. Abgesehen davon, das er keine Kind-Elemente enthalten kann und nur auf ein DokumentItem verweist, ist er mit dem ElementNode-Knoten identisch.

Der *TextNode*-Knoten verweist auf einen oder mehrere DocumentItem Objekte, die den Typ Text haben. Er repräsentiert den Text-Inhalt, auch "Character Data" genannt, eines Elements. Wie beim DOM sind keine nebeneinander liegenden TextNode-Knoten erlaubt. Zwischen ihnen muss immer mindestens ein ElementNode stehen.

Je nachdem, wie die verschiedenen Annotationsebenen annotiert sind, muss

[3]Eigentlich könnte den Segmenten vom Typ Ignore eine Annotationsebenen-Id "ignore" zuweisen werden, jedoch für sie irrelevant, auf welcher Ebene sie existieren und der Einfacherheit wird ihnen deswegen der Wert "every" zugewiesen.

ein TextNode-Knoten auf mehrere DokumentItem-Objekte verweisen, da ein Text-Inhalt auf der einen Ebene durch Elemente auf einer anderen Ebene aufgetrennt wird. Das kurze Fragment aus der Annotation des Gedichts illustriert diese Vorgehensweise, die auch *Slicing* genannt wird.

```
<(1)l><(2)s>Sein Blick ist vom Vorübergehn der Stäbe</(1)l>
<(1)l>so müd geworden, dass er nichts mehr hält.</(2)s></(1)l>
```

Auf der Annotationsebene "1" wird ein Satz annotiert, der über zwei Zeilen geht. Auf der Annotationsebene "2" befindet sich die Annotation für den Satz. Im hierarchischen Modell für die Annotationsebene "1" wird der Satz in zwei Teilen annotiert und es müssen zwei TextNode-Knoten eingefügt werden, die jeweils auf eines der Segmente des Satzes verweisen und jeweils ein "`l`" ElementNode-Knoten als Elternknoten haben.

Im Gegensatz dazu gibt es auf der Annotationsebene "2" nur einen TextNode-Knoten, der auf die beiden Segmente verweist und eine "`s`" ElementNode-Knoten als Elternknoten hat.

Die einzelnen hierarchischen Strukturen sind unabhänging voneinander und gleiche TextNode-Knoten werden nicht zwischen ihnen geteilt .

Der Wert eines TextNode-Knotens entspricht den Inhalten der DocumentItems, auf die er verweist. Wenn er auf mehrere DocumentItems verweist, werden die einzelnen Werte in der Reihenfolge, in der sie im Dokument stehen, aneinander gehängt. Betrachtet man das vorherige Beispiel, ist der Wert des TextNode-Knotens auf Annotationsebene "2" die Zeichenkette "Sein Blick ist vom Vorübergehn der Stäbe so müd geworden, dass er nichts mehr hält.".

Die Abbildung 4.1 zeigt beispielhaft eine Instanz des Datenmodells, die an die Annotation des Gedichts angelehnt ist. Im unteren Teil der Abbildung ist das Dokument illustriert. Es ist als Kette von einzelen Zeichen abgebildet. Darunter ist die Position der Zeichen innerhalb des Dokuments angegeben.

Die Annotation besteht aus fünf Segmenten: zwei Start-Elementen, zwei End-Elementen und einer Textregion. Die lineare Struktur des Modells wird durch fünf DocumentItem-Objekte gebildet. Ihr Offset-Wert gibt jeweils die Start-Position des Segments an, der Length-Wert ihre Länge.

Oberhalb der gestrichelten Linie ist die hierarchische Struktur abgebildet. Jede Annotationsebene besteht aus einem AnnotationsLayer-, einem ElementNode- und einem TextNode-Objekt. Die Linien bei den ElementNode- und TextNode-Objekten geben an, auf welches Segment sie verweisen.

Gegenüber der Beschreibung in [12] hat sich die Struktur des Verarbeitungsmodells geändert. Dort wurde eine Spezialisierung der linearen Struktur in Start-, End-, Empty- und Text-Objekte vorgenommen. Für die Implementierung hat sich dies jedoch als unperformant erwiesen, daher wurde die Modellierung entsprechend durch das generische DocumentItem mit einer Typangabe modifiziert.

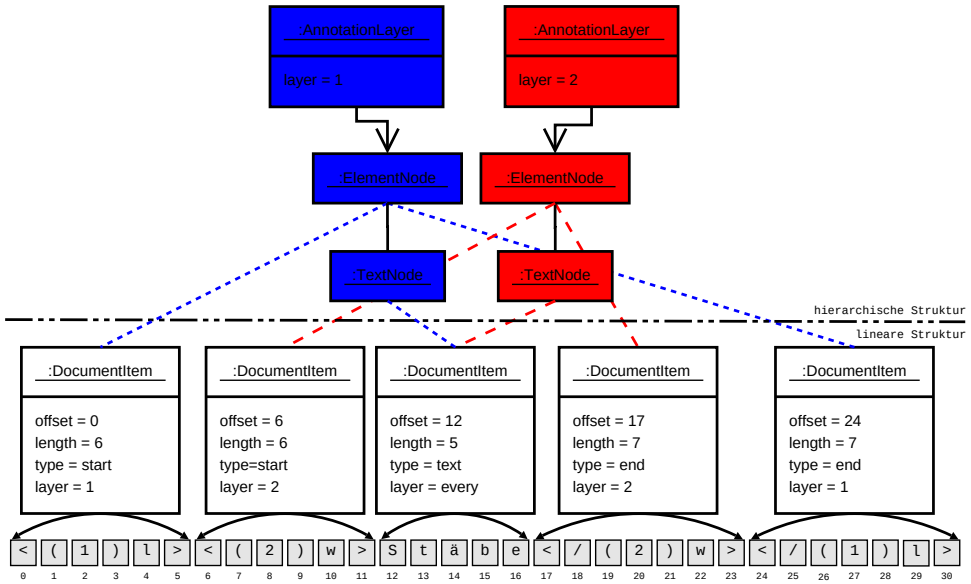


Abbildung 4.1: Eine Instanz des Datenmodells

## 4.3 Vergleich der Verarbeitungsmodelle

Die beiden Verarbeitungsmodelle haben jeweils eine andere Herangehensweise an das Problem der Modellierung eines multi-hierarchisch strukturierten Dokuments.

Das Modell von Hilbert rückt das Konzept der Just-In-Time-Trees in den Mittelpunkt. Dies bedeutet jedoch, das für ein Dokument mit $n$ Annotationsebenen $n$ Instanzen des Verarbeitungsmodells benötigt werden, um alle Ebenen gleichzeitig darzustellen. Aus der Sicht von Hilbert arbeitet der Benutzer jedoch immer zu einem Zeitpunkt auf einer Annotationsebene. Durch einen Parser wird aus dem Dokument eine Instanz des Verarbeitungsmodells mit einer bestimmten Ebene als primäre Annotationsebene erzeugt. Das Annotationswerkzeug kann dem Benutzer beispielsweise eine Ansicht der Baum-Struktur präsentieren.

Müssen jedoch Operationen ausgeführt werden, die die Instanzierung aller Annotationsebenen vorrausetzten, müssen entweder alle $n$ Instanzen generiert werden oder eine bestehende Instanz muss entsprechend umgeformt werden. Diese Umformung ist eine aufwendige Aufgabe. Solche Operationen sind beispielsweise das Validieren des Dokuments oder das Exportieren der einzelnen Annotationsebenen in jeweils eine Datei.

Im Gegensatz dazu existieren alle hierarchischen Strukturen der Annotationsebenen beim dem hier vorgestellten Ansatz parallel zueinander. Alle Ebenen sind gleichberechtigt. Ein erneutes Parsen oder die Umformung der

bestehenden Struktur ist nicht notwendig, wenn eine andere Annotationsebene betrachtet werden soll. Das Modell ist als Datenspeicher für die Implementierung eines Annotationswerkzeugs konzipiert und die lineare Struktur wird bereits bei den Änderungen, die der Benutzer vornimmt, geändert; ein erneutes Parsen ist nicht notwendig, um sie zu erstellen.

Hilbert setzt den Focus auf eine hierarchische Struktur. Durch die Foreign-Elemente können auch Relationen zwischen einzelnen Elementen verschiedener Annotationsebenen untersucht werden. Man kann beispielsweise nachsehen, ob ein bestimmtes Element der aktuellen Annotationsebene von bestimmten Elementen der anderen eingeschlossen wird.

Der hier vorgestellte Ansatz rückt die Linearisierung des annotierten Dokuments in den Mittelpunkt. Dies ermöglicht es Modifikationen der Annotation, beispielsweise durch Operationen, die ein Benutzer in einem Annotationswerkzeug vornimmt, einfach auf das Datenmodell umzusetzen. Die Datenstruktur kann durch Einfügen oder Löschen bzw. Verlängern oder Verkürzen eines Segments effizient geändert werden und stellt in Echtzeit ein konsistentes Modell zur Verfügung.

Weiterhin sind alle Annotationsebenen gleichberechtigt und existieren unabhängig voneinander. Eine Validierung eines Dokuments kann so schnell erfolgen und ein Export der einzelnen Annotationsebenen in verschiedene Dateien lässt sich so leicht realisieren.

Je nach Fragestellung kann diese Unabhängigkeit sich jedoch auch als hinderlich erweisen, da es nicht möglich ist, Elemente verschiedener Annotationsebenen zueinander in Relation zun setzen. In wie weit beispielsweise ein Element einer Ebene Elemente andere Ebenen einschließt kann anhand der hierarchischen Struktur nicht untersucht werden.

# Kapitel 5

# Implementierung

Der Prototyp eines Annotationswerkzeugs, im folgenden auch Editor genannt, ist mit der Programmiersparche C++ ([18]) unter Verwendung des wxWidgets-Toolkits erstellt worden.

wxWidgets ist eine C++ Bibliothek, die das Erstellen von plattformunabhängigen graphischen Benutzeroberflächen erleichtert. Es ist unter anderem für Windows, Mac OS X als auch für diverse Unices verfügbar. Die Verwendung der wxWidgets-Bibliothek bietet dem Entwickler den Vorteil, plattformübergreifend das gleiche Application Programming Interface (API) zu verwendenden und so die Applikationen ohne wesentlichen Aufwand auf verschiedenen Betriebssystemen zur Verfügung stellen zu können. Andererseits erfolgt die Darstellung der Anwendung für den Benutzer systemabhängig in einer ihm bereits bekannten Art und Weise, so dass das notwendige Erlernen des Umgangs mit grundlegenden Funktionen des Programms entfällt. Eine wxWidgets-Anwendungen sieht beispielsweise unter Windows wie eine normale Windows-Anwendung aus und unter Unix wie eine GTK-Anwendung[1].

Weiterhin hat die gute Dokumentation als auch die einfache Integration in die Microsoft Visual C++ Entwicklungsumgebung und GNU-Toolchain zur Entscheidungsfindung beigetragen. Sofern von der Anwendung keine speziellen Betriebssystem-Routinen verwendet werden, sind nur minimale Änderungen am Quellcode notwendig, um das Programm auf ein anderes Betriebssystem zu portieren, da wxWidgets eine sehr gute Abstraktion zum Betriebssystem und dessen nativen Toolkits bietet. In [15] wird eine sehr gute Einführung in wxWidgets gegeben.

Neben Microsoft Visual C++ für die Entwicklung unter Windows wurde unter den Unices die GNU-Toolchain verwendet. Die populären Programme Make, Autoconf, Automake und Libtool, die unter fast allen Unices verfügbar

[1] Das GIMP-Tool-Kit (GTK) ist eine unter Unix und Unix-Derivaten populäre C-Bibliothek zum Erstellen graphischer Oberflächen. Das GNOME Projekt setzt GTK beispielsweise als Grundlage für ihren Desktop ein.

sind, gewährleisten einen portablen und standardisierten Build-Prozess zum Übersetzen und Installieren der Software aus den Quelldateien.

Die Entscheidung für die Verwendung von C++ im Gegensatz zu Java ist dadurch begünstigt worden, dass C++ schneller ist und im Gegensatz zu Java keine Laufzeitumgebung, die nicht immer auf jedem System vorhanden ist, braucht.

Im Folgenden wird als nächstes die Implementierung des Verarbeitungsmodells betrachtet und danach das genauer die Annotationswerkzeug genauer beschrieben. Beim Verarbeitungsmodell werden die beiden Hauptkomponenten, das Datenmodell und der Editor, jeweils getrennt beschrieben.

## 5.1 Datenmodell

Ein Dokument wird, wie in Abschnitt 4.2.1 beschrieben wird, in einzelne Segmente geteilt, die entweder das Markup oder die Text-Inhalte repräsentieren. Die einzelnen Segmente werden über einer Sequenz von Zeichen gebildet und haben einen Start-Punkt und eine Länge. Die lineare Struktur des Datenmodells ist eine Liste von Segmenten, also die Aneinanderreihung der einzelnen Segmente in der Reihenfolge ihrer Startpositionen.

Durch einen Parser wird zusätzlich zur linearen Struktur für jede Annotationsebene eine hierarchische Struktur aufgebaut. Die Abbildung 5.1 zeigt ein UML-Diagramm der Implementierung des Datenmodells.

### 5.1.1 Hauptklassen

Die Hauptkomponente des Datenmodells ist die Klasse `MultiLayerDocument`. Diese wird, wie die meisten Klassen, die zum Datenmodell gehören, in der Header-Datei `Document.hpp` deklariert. Als Containerklasse speichert sie die Annotationsebenen, die lineare Struktur und die hierarchische Struktur. Sie bietet entsprechende Zugriffsmethoden an, um die Inhalte zu modifizieren.

Durch die Methode `GetLayerCount` kann die Anzahl der Annotationsebenen ermittelt werden und die Methode `CreateLayer` erzeugt eine neue Ebene. Die `GetLayer`-Methoden, die in einer Version eine Layer-Id oder eine Annotationsebenen-Id als Argument erwarten, erlauben den Zugriff auf die AnnotationLayer-Objekte.

Mit `GetCount` kann die Anzahl der Segmente in der linearen Struktur ermittelt werden und die Methoden `GetItem` und `GetLastItem` ermöglichen einen Zugriff auf die Objekte. Mit `Insert` und `Delete` können neue Segmente eingefügt oder gelöscht werden.

Das MultiLayerDocument-Objekt kann Nachrichten an andere Objekte versenden, wenn sich sein Zustand verändert. Die Methoden `AddDocument-`

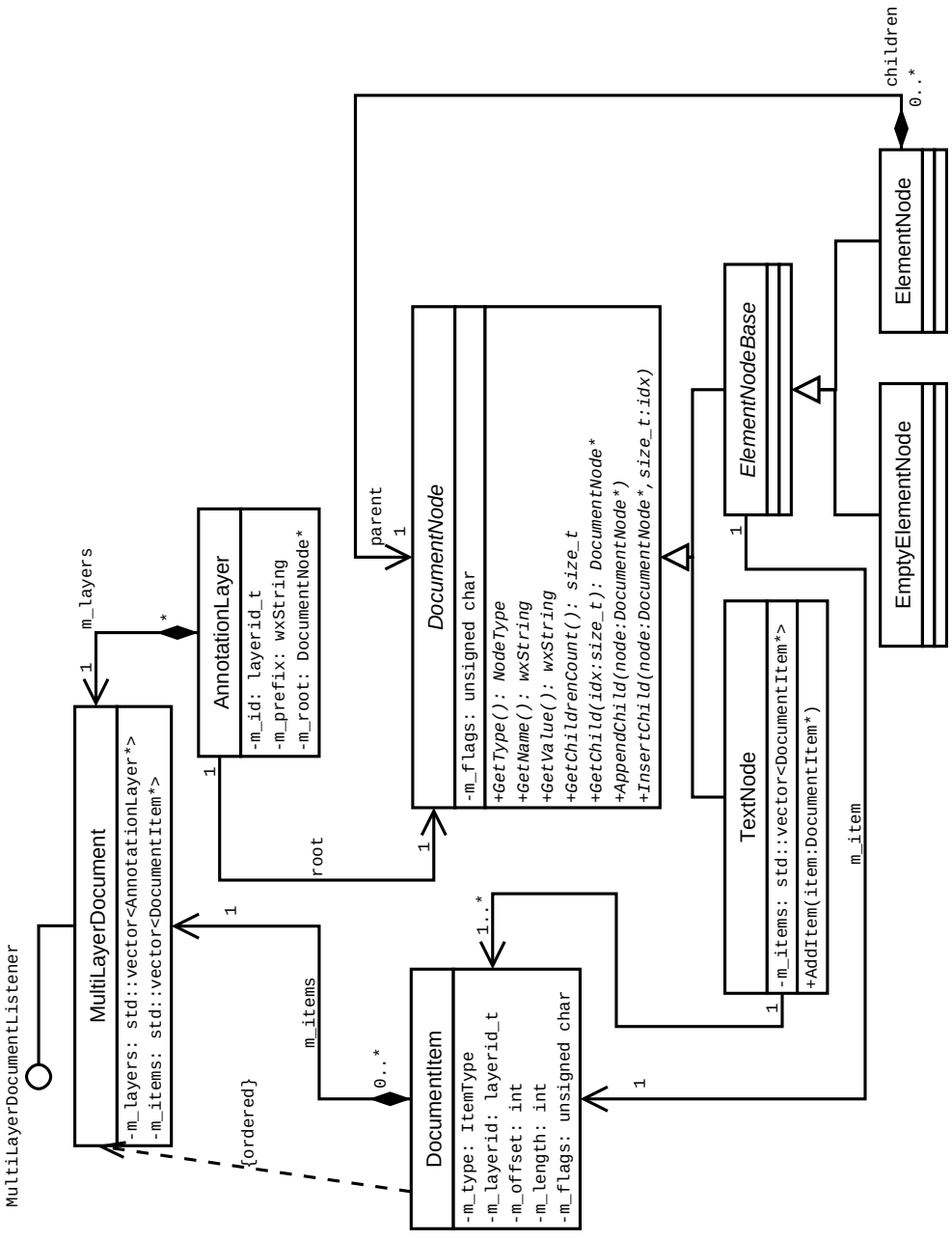


Abbildung 5.1: UML-Diagramm des Datenmodells

`Listener` und `RemoveDocumentListener` ermöglichen entsprechenden Nachrichtensenken hinzuzufügen oder entfernen.

Sie implementiert einen Container und Zugriffsmethoden auf die lineare und hierarchische Struktur des Datenmodells. Die eigentlichen Daten, d. h. die einzelnen Zeichen, die ein Dokument ausmachen, werden nicht in dieser Klasse gespeichert. Die abstrakte Klasse `DataAccess` ermöglicht durch die Methode `ReadRange` unter Angabe eines Start- und Endoffsets des Zugriff auf die Zeichen. Diese Methode muss jeweils für die entsprechende Anwendung implementiert werden. Durch Verwendung dieser abstrakten Klasse wird einerseits der Speicher, der für die Speicherung des Dokuments benötigt wird, reduziert und andererseits das Datenmodell von der Implementierung des Annotationswerkzeugs entkoppelt.

Die Annotationsebenen werden durch die Klasse `AnnotationLayer` implementiert und vom MultiLayerDocument verwaltet. Jedes AnnotationLayer-Objekt hat eine Layer-Id, einen eindeutigen Wert zwischen $-2^{31}-1$ und $2^{31}-1$, der die Annotationsebene identifiziert. Die Zeichenketten der Annotationsebenen-Ids werden so auf einen nummerischen Wert abgebildet, der es anderen Objekten erleichtert, auf eine Annotationsebene zu verweisen ohne Pointer verwenden zu müssen. Sollte eine Annotationsebene gelöscht werden, und ein anderes Objekt noch einen Pointer auf diese Ebene benutzt und dereferenziert, also versucht, den Pointer zu verfolgen, kommt das Programm in einen undefinierten Zustand.

Die Layer-Id wird einer Annotationsebene automatisch bei der Erstellung zugewiesen und kann nicht verändert werden. Negative Layer-Id Werte werden an Annotationsebenen vergeben, die automatisch vom Programm erzeugt werden, wenn eine Annotationsebenen-Id gefunden wird, zu dem noch keine Ebene existiert. Diese auch "auto-created" genannten Ebenen werden automatisch gelöscht, wenn sie kein Markup mehr enthalten. Durch die Methode `IsAutoCreated` kann abgefragt werden, ob die Annotationsebene automatisch erstellt wurde. In diesem Fall liefert die Methode den Wert `true` zurück.

Zwei Werte sind reserviert und werden einer Annotationsebene nicht zugewiesen, sondern intern verwendet. Zum einen `LAYERID_INVALID` ($-1$), der eine ungültige Ebene beschreibt. Zum anderen `LAYERID_EVERY_LAYER` (0), der jede vorhandene Ebene bezeichnet.

Weiterhin speichert die Klasse AnnotationLayer noch die Annotationsebenen-Id. Dies ist die Zeichenkette, die bei den Elementen in den Runden angegeben ist. Die Annotationsebenen-Id kann durch die Methode `GetPrefix` abgefragt werden und durch die Methode `SetPrefix` gesetzt werden. Zur Zeit wird beim Setzen des Werts keine Prüfung durchgeführt, ob die Annotationsebenen-Id eventuell noch für andere Annotationsebenen-Objekte vergeben ist. Dieses Verhalten muss noch implementiert werden.

Die Validität einer Annotationsebene lässt sich durch die Methode `GetValidity` abfragen. Einer der Werte `VALIDITY_UNKNOWN` (unbekannt), `VALIDI-`

`TY_INVALID` (nicht valide), `VALIDITY_WELLFORMED` (wohlgeformt) oder `VALIDITY_VALID` (valide) wird zurückgegeben.

Jedes Annotationsebenen-Objekt besitzt eine so genannte "ParsedGeneration". Dieser Wert, der mit der Methode `GetParsedGeneration` abgefragt werden kann, ändert sich jedesmal wenn die Annotationsebene neu geparst wird. So kann beispielsweise die Struktur in einem Annotationswerkzeug angezeigt werden und muss nur aktualisiert werden, wenn sich die "ParsedGeneration" geändert hat.

Wenn der Parser eine hierarchische Struktur zu einer Annotationsebene aufbauen kann, d. h. sie mindestens wohlgeformt ist, kann mit der Methode `GetRoot` auf den Wurzel-Knoten der Ebene zugegriffen werden. Sie liefert ein Objekt vom Typ DocumentNode oder den Wert `null`, wenn keine Ebene existiert, zurück.

### 5.1.2 Klassen der linearen Struktur

Die einzelnen Segmente, also die Elemente der linearen Struktur, werden von der Klasse `DocumentItem` implementiert. Sie speichert den Startpunkt und die Länge des Segments. Diese Werte werden in Byte angegeben, da wxWidgets in den verwendeten Komponenten mit Byte-Offsets arbeitet und so ein Umrechnen auf einzelne Zeichen nur unötigen Aufwand bedeuten würde. Weiterhin muss so auch die Unicode-Codierung nicht normalisiert werden.

Jedes DocumentItem besitzt eine Layer-Id, die die Annotationsebene referenziert, auf der es sich befindet. Wenn ein DocumentItem kein Teil das Markup ist, also nur einen Text-Inhalt beschreibt, wird der Wert LAYERID_EVERY_LAYER verwendet, da es sich dann auf allen Ebenen befindet. Sollte einem DocumentItem keine Annotationsebene zugewiesen werden können, weil beispielsweise die Syntax für ein Element nicht korrekt ist, wird der Wert LAYERID_INVALID verwendet. Der Wert kann mit der Methode `GetLayerId` abgefragt werden.

Jedes DocumentItem bekommt einen Typ zugeordnet. Er wird im Falle des Annotationswerkzeugs bei der Benutzereingabe bestimmt. Je nachdem, in welcher Reihenfolge der Benutzer die Zeichen eingibt bzw. editiert, muss sich der Typ ändern. Die Klasse MultiLayerDocument stellt die Methode `ClassifyStringData` bereit, die zu einer gegebenen Zeichenkette heuristisch einen Typ berechnet. Folgende Typen sind definiert:

`ITEM_TEXT` Text-Inhalt; das Segment enthält kein Markup.

`ITEM_ELEMENT_START` Dieses Segment enthält ein Start-Tag.

`ITEM_ELEMENT_END` Dieses Segment enthält ein End-Tag.

`ITEM_ELEMENT_EMPTY` Dieses Segment enthält ein leeres Element.

`ITEM_IGNORE` Dieses Segment soll vom Parser ignoriert werden. Zur Zeit wird dieser Wert für die XMC-Deklaration und die Annotationsschema-Deklarationen verwendet, da diese noch nicht implementiert worden sind.

Die DocumentItem Objekte werden, nach ihren Start-Offsets sortiert, als Liste vom MultiLayerDocument gespeichert. Weiterhin werden noch Attribute vom DocumentItem gespeichert. Die Modellierung erlaubt dabei, dass auch End- und Text-Segmente Attribute tragen; die Implementierung verbietet dies jedoch. Eine andere Modellierung, die das nicht erlauben würde wurde aus Effizienzgründen verworfen, da sie eine Spezialisierung der Klasse DocumentItem für Objekte mit und ohne Attribut erfordert hätte. Der Typ eines Objekts ist jedoch abhängig von den Eingaben des Benutzers. Je nachdem wie welche Modifikationsoperationen durch den Benutzer vorgenommen werden, kann sich der Typ beispielsweise von `ITEN_ELEMENT_START` nach `ITEN_TEXT` und wieder zurück ändern. Bei jedem Übergang von einem zu einem anderen Typ müsste eine neue Instanz der entsprechenden DocumentItem-Objekts erstellt werden und dies erzeugt einen nicht zu unterschätzenden Overhead.

Weiterhin besitzt jedes DocumentItem eine Reihe von Flags, die seinen internen Status beschreiben. Diese Flags werden bislang ausschliesslich durch den Parser verwendet. Um Speicherplatz zu sparen sind die einzelnen in einem C++-Enum-Typen realisiert und werden in einem Objekt-Attribut vom Datentyp "`unsigned char`" gespeichert. Durch binäre Operationen werden die einzelnen Bits, die jeweils einem Flag-Wert entsprechen angefragt, gesetzt oder gelöscht. Folgende Flag-Werte sind definiert und können in beliebiger Kombination verwendet werden:

`FLAG_PARSED` Das Segment wurde vom Parser verarbeitet.

`FLAG_PARSED_OK` Das Segment wurde vom Parser als gültig befunden.

`FLAG_WHITESPACE_ONLY` Das Segment enthält nur Whitespaces.

Mit der Methode `GetFlags` kann der Wert der Flags abgefragt werden. Gesetzt werden sie ausschliesslich vom Parser.

Die Start-Position und die Länge des Segments kann durch die Methoden `GetOffset` und `GetLength` angefragt werden. Mit der Methode `IsEmpty` kann festgestellt werden, ob die Länge einen Wert grösser 0 hat. Mit `GetEndOffset` kann die Endposition bestimmt werden. Dieser Wert wird aus Start-Position und Länge berechnet. Die `IsInside` liefert den Wert `true` zurück, wenn die Position, die als Argument übergeben wird, innerhalb des Segments liegt.

Die initiale Start-Position und Länge werden dem Konstruktor des Objekt übergeben. Dieser ist jedoch nicht öffentlich. Er wird automatisch beim Einfügen von neuen Segmenten aufgerufen.

Durch die Methoden `Move`, `SetLength`, `IncLength` und `DecLength` können die Start-Position oder die Länge verändert werden.

Mit `Move` wird das Segment um einen bestimmten Wert verschoben. Die neue Start-Position muss grösser oder gleich 0 sein. Mit den Methoden `IncLength` und `DecLength` kann ein bestimmter Wert zu der aktuellen Länge addiert oder davon substrahiert werden. Mit `SetLength` kann der Länge direkt ein Wert zugewiesen werden. Wie auch die Start-Position muss der Wert der Länge grösser oder gleich 0 sein.

Auf den Inhalt, der von einem Segment eingeschlossen wird, kann mit der Methode `GetData` zugegriffen werden. Sie lief eine Zeichenkette zurück, die bei einem Text-Segment den Primärdaten entspricht. Bei den anderen Segmenten entspricht sie dem Markup.

Weiterhin werden noch die Methoden `GetName` und `GetValue` zur Verfügung gestellt. Nachdem der Parser das Segment bearbeitet hat, liefert die Methode `GetName` bei Segmenten vom Typ `ITEM_ELEMENT_START`, `ITEM_ELEMENT_END` oder `ITEM_ELEMENT_EMPTY` den Namen des Elements zurück, bei Segmenten vom Typ `ITEM_TEXT` wird die Zeichenkette "`#text`" zurückgegeben. Die Methode `GetValue` liefert bei Segmenten vom Typ `ITEM_TEXT` den Text-Inhalt des Segments und bei den anderen Typen den Wert, den die Methode `GetData` zurückliefert.

Jedem DocumentItem wird eine eindeutige Id zugewiesen. Über die Methode `GetId` kann sie abgefragt werden. Sie kann verwendet werden, um auf DocumentItem-Objekte zu verweisen, ohne Pointer nutzen zu müssen.

### 5.1.3 Klassen der hierarchischen Struktur

Die hierarchische Struktur der einzelnen Annotationsebenen wird durch eine Baumstruktur gebildet. Daher wurde das Design-Pattern "Kompositum" ([9]) wurde für die Implementierung eingesetzt. Die Rolle der Komponente übernimmt die abstrakte Klasse `DocumentItem`. Die Blätter werden durch die Klassen `TextNode` und `EmptyElementNode` modelliert und die Klasse `ElementNode` bildet das Kompositum. Die Klassen `ElementNode` und `EmptyElementNode` sind konkrete Realisierungen der abstrakte Klasse `ElementNodeBase`. Diese verwaltet jedoch nur einen Pointer auf ein DocumentItem vom Typ `ITEM_ELEMENT_START` beziehungswiese `ITEM_ELEMENT_EMPTY`. Bei einer Überarbeitung des Datenmodells sollten die Klassen `ElementNodeBase` und `EmptyElementNode` wegfallen. Erstere ist nur eingeführt worden, um eine Duplizierung von Code in den Klassen `EmptyElementNode` und `ElementNode` zu vermeiden. Letzere ist eigentlich nicht notwendig, da ihre Funktion durch Erweiterung der Klasse `ElementNode` eleganter zu realisieren wäre.

Die hierarchische Struktur wird, im Gegensatz zu der linearen Struktur, nicht synchron mit den Benutzeränderungen im Editor aufgebaut. Wenn der Benutzer eine gewisse Zeit[2] keine Modifikationen vornimmt, wird diese Struk-

[2]In der Prototypenimplementierung 2 Sekunden.

tur von einem Parser erstellt. Im Abschnitt 5.2 wird näher auf den Parser eingegangen.

Als abstrakte Klasse kann die Klasse `DocumentNode` nicht instanziert werden, sondern stellt die Schnittstelle für anderer Programmteile dar, die die hierarchische Struktur verarbeiten sollen. Die folgenden Methoden und deren Eigenschaften gelten für alle von `DocumentNode` abgeleiteten Klassen. Mit der Methode `GetType` kann der Typ eines Objekts festgestellt werden. Es sind folgende Werte definiert:

`NODE_TEXT` Dieser Knoten ist ein Text-Knoten.

`NODE_ELEMENT` Dieser Knoten ist ein Element-Knoten.

`NODE_ELEMENT_EMPTY` Dieser Knoten ist ein Element-Knoten, der keine weiteren Kind-Knoten enthält.

Die Methode `GetChildrenCount` ermittelt die Anzahl der Kind-Knoten, die ein Knoten enthält. Bei TextNode- und EmptyElementNode-Knoten ist der Rückgabewert stets "0". Mit der Methode `GetChild` kann ein Knoten aus der Liste der Kinder eines Knoten ausgewählt werden. Die Methode erwartet als Argument einen Index und liefert den entsprechenden Knoten zurück. Ist der Index in der Liste nicht gültig oder kann der Knoten keine Kinder enthalten wird der Wert `null` zurückgegeben.

Die Methode `AppendChild` erlaubt das anhängen eines Kindknotens und die `InsertChild` fügt den Kind-Knoten an die angegeben Position ein. Diese Methoden sind jedoch nur für die Klasse `ElementNode` implementiert, bei den anderen Klassen wird der Aufruf ignoriert, da sie keine Kind-Element enthalten können.

Die Klasse TextNode realisiert Text-Knoten in der hierarchischen Struktur. Jeder Text-Knoten verweist auf mindestens ein DocumentItem-Objekt vom Typ `ITEM_TEXT`. Er kann auf mehrere DocumentItem-Objekte verweisen, da ein Text-Segment einer Annotationsebene durch eine andere Ebene fragmentiert werden kann. In Abschnitt 4.2.2 wird das so genannte "Slicing" erläutert. Durch die Methode `AddItem` können einem TextNode-Objekt mehrere DocumentNode-Objekte vom Typ `ITEN_TEXT` zugewiesen werden.

Die Methode `GetName` liefert immer die Zeichenkette "`#text`" zurück. Auf den Text-Inhalt des Knoten kann man mit der Methode `GetValue` zugreifen. Wenn der Text-Knoten auf mehrere DocumentItems verweist, werden die Text-Inhalte zusammengesetzt und dann zurückgegeben.

Die Klassen `ElementNode` und `EmptyElementNode` liefern beim Aufruf der Methode `GetName` den Namen des Elements; die Methode `GetValue` liefert das Tag in Rohform, d. h. die Zeichenkette, die im Annotationswerkzeug eingegeben wurde, zurück.

## 5.2 Parser

Der Parser erstellt aus der linearen Struktur für jede Annotationsebene eine hierarchische Struktur. Dabei werden alle Annotationsebenen gleichzeitig verarbeitet. In der aktuellen Implementierung des Annotationswerkzeugs wird der Parser nach 2 Sekunden aufgerufen, wenn der Benutzer eine Änderung am Dokument vorgenommen hat.

Die Verarbeitung findet in zwei Schritten statt, die jedoch gekoppelt sind, so dass die Segmente nicht zweimal durchlaufen werden müssen.

Bei der Initialisierung des Parsers wird für jede Annotationsebene ein Status-Objekt intialisiert, das einen Stack enthält. Er erlaubt die üblichen Operationen `Pop` (ein Element auf dem Stack ablegen), `Push` (das oberste Element vom Stack nehmen) und `Peek` (das oberste Element vom Stack ansehen). Weiterhin kann mit der Operation `PeekAt` auch ein beliebiges Element vom Stack betrachtet werden.

Die Segmente werden, beim Ersten beginnend, nacheinander abgearbeitet. Als erstes muss der Parser die einzelnen Segmente analysieren. Danach kann mit den Segmenten die hierarchische Struktur aufgebaut werden.

Der Parser wird durch die Klasse `MultiLayerDocumentParser` implementiert, die in der Datei `MultiLayerDocumentParser.hpp` deklariert wird.

### 5.2.1 Analyse der linearen Struktur

Der Parser kann durch das Abfragen des `FLAG_PARSED`-Flags des DocumentItems, das das Segment bildet, überprüfen, ob es, im Bezug auf die lineare Struktur, verarbeitet werden muss. Ist dieses Flag nicht gesetzt, muss der Parser das Segment analysieren. Wenn der Parser das Segment verarbeitet hat, wird das Flag gesetzt. Wenn der Benutzer im Annotationswerkzeug das Dokument bearbeitet, wird dieses Flag für die Segmente, die von den Änderungen betroffen waren, gelöscht. Dieses Verfahren hat den Vorteil, dass immer nur die Segmente verarbeitet werden müssen, die verändert wurden[3].

Dieser Verarbeitungsschritt fügt den DocumentItem-Objekten, die die Datenstruktur der Segmente bilden, weitere Informationen hinzu. Nach der heuristischen Bestimmung des Typs bei der Eingabe wird nun eine genauere Untersuchung des Segments vorgenommen.

Die Methode `ParseItem` der Klasse `MultiLayerDocumentParser` implementiert die Analyse der Segmente der linearen Struktur. Dazu wird ein Segment-Parser verwendet, der durch einen endlichen Automaten mit 17 Zuständen realisiert ist.

[3]Dies gilt natürlich nur für die Elemente der linearen Struktur; in der aktuellen Implementierung wird die hierarchische Struktur immer komplett neu aufgebaut.

Der Segment-Parser kann die einzelnen Segmente in Text, Start-Tag, End-Tag und Empty-Tag klassifizieren. Die Unterstützung für Annotationschema-Deklarationen und globale Kommentare oder Verarbeitungsanweisungen ist bislang nur rudimentär implementiert. Sobald der Segment-Parser die Zeichen "`<`" gefolgt von "`?`" oder "`!`" verarbeitet hat springt er in den Zustand `ACCEPT` und beendet die Verarbeitung. Dies passiert unabhängig davon, ob eine Anweisung beendet ist oder nicht. Der Typ des DocumentItems wird in diesen Fällen auf `ITEM_IGNORE` gesetzt. Wenn der Parser auf einen lokalen Kommentar oder eine lokale Verarbeitungsanweisung trifft, also solche, denen jeweils eine Annotationsebenen-Id vorangestellt ist, meldet er einen Fehler.

Bei der Analyse von Segmenten, die als Text klassifiziert werden, kann der Segment-Parser feststellen, ob der Text nur Whitespaces enthält. Wenn dies der Fall ist, wird das Flag `FLAG_WHITESPACE_ONLY` für das entsprechende DocumentItem gesetzt.

Bei der Verarbeitung der hierarchischen Struktur ist eine Unterscheidung zwischen Text-Segmenten, die nur aus Whitespaces bestehen und anderen notwendig, damit diese beim Aufbau der hierarchischen Struktur gegebenenfalls ignoriert werden können. Das folgende Beispiel soll diese Notwendigkeit illustrieren. Zeilenumbrüche werden als Zeichenfolge "`\n`" dargestellt.

```
<!DOCTYPE (1)lg SYSTEM "teivers2.dtd">\n
<!DOCTYPE (2)text SYSTEM "teiana2.dtd">\n
<(1)lg>\n
[..]
```

Für jeden Zeilenumbruch wird in der linearen Struktur ein separates Segment erstellt, da es zwischen anderen (Non-Text-)Segmenten liegt. Die ersten zwei Segmente liegen jedoch ausserhalb des Wurzelelements der ersten Annotationsebene und würden so das Wohlgeformtheitskriterium der Ebene verletzen, da keine Elemente ausserhalb des Wurzelknotens liegen dürfen. Beim Aufbau der hierarchischen Struktur können diese Segmente jedoch ignoriert werden, da sie nur Whitespaces enthalten und allein zur Formatierung der Annotation eingesetzt werden.

Bei der Analyse von Start-, End- und Empty-Tags wird die Syntax des entsprechenden Segments überprüft und das Tag in seine einzelnen Komponenten zerlegt. Diese Komponenten, die die Annotationsebenen-Id, den Element-Namen, die Attribut-Namen und deren Attribut-Werte umfassen, werden auf ihre Schreibweise geprüft. Je nach Komponente können nur bestimmte Zeichen verwendet werden. Sollte ein Zeichen gefunden werden, das nicht erlaubt ist, gelangt der Automat für die Analyse in den Fehlerzustand `REJECT`.

Sollte kein Fehler festgestellt worden sein, werden der Typ, der geparste Element-Name, die Attribute und deren Werte[4] im DocumentItem-Objekt

[4] In der aktuellen Implementierung werden Attribute zwar extrahiert aber noch nicht dem DocumentItem zugewiesen.

gesetzt, so dass auf diese Werte zugegriffen werden kann, ohne das Segment erneut parsen zu müssen..

Die Annotationsebenen-Id wird in eine nummerische Layer-Id umgewandelt. Wird eine Annotationsebenen-Id gefunden, zu der noch keine Layer-Id existiert und damit auch noch kein AnnotationLayer-Objekt vorhanden ist, wird eine neue Ebene angelegt. Diese Annotationsebene wird als "auto-created" angelegt, da sie vom Parser erstellt wird. Wenn nach dem Parsen keine Elemente mehr auf einer "auto-created" Annotationsebene vorhanden sind, wird diese Ebene automatisch gelöscht.

In einer zukünfigen Implementierung sollte der Parser die Annotationsschema-Deklarationen auswerten und automatisch Annotationsebenen anlegen. Da diese jedoch deklariert sind, sollte sie dann natürlich nicht als "auto-created" angelegt werden.

Wenn nach dem Verarbeiten des Segments der Automat in den Endzustand `ACCEPT` kommt, konnte beim Parsen kein Fehler festgestellt werden und das Segment ist syntaktisch korrekt. In diesem Fall wird das Flag `FLAG_PARSED_OK` gesetzt. Wenn der Automat nicht in einen akzeptierenden Zustand kommt, wird das Segment als fehlerhaft markiert und bekommt die Layer-Id `LAYERID_INVALID` zugewiesen.

Unabhängig davon in welchem (End-)Zustand sich der Automat nach der Verarbeitung dies Segments befindet, wird das Flag `FLAG_PARSED` gesetzt. Dieses Flag wird nur zurückgesetzt, wenn eine Änderung an dem Segment vom Benutzer vorgenommen wurde. So kann der Parser die Analyse des Segments zwischen zwei Aufrufen des Parsers überspringen, wenn es nicht verändert wurde.

### 5.2.2 Aufbau der hierarchischen Strukturen

Nachdem ein Segment geparst worden ist, kann es zum Aufbau der hierarchischen Strukturen verwendet werden. Für jede Annotationsebene, auch solche die während des Parsens als "auto-created" angelegt werden, wird im Parser ein, inital leerer, Element-Stack erzeugt. Alle Annotationsebenen werden gleichzeitig aufgebaut.

Aus den verschiedenen Segmenten werden die Knoten der hierarchischen Struktur aufgebaut. Dies sind in der aktuellen Implementierung Element-Knoten und Text-Knoten.

Die Abbildung 5.2 zeigt den Pseudo-Code des Parsers. Der Code für die Analyse der linearen Struktur eines Segments und die Verarbeitung von Whitespaces, die ausserhalb der Wurzelelemente der Annotationsebenen liegen, ist nicht aufgeführt.

Der Aufbau der hierarchischen Struktur erfolgt durch das Ablegen, Entfernen und Vereinigen der Elemente auf dem entsprechenden Stack der Annotationsebene. Die Segmente werden nacheinander verarbeitet und je nachdem,

```
/* initalisierung der Parser-States für alle Annotationsebenen */
for each item in DocumentItemList do
  if not(IsFlag(item, FLAG_PARSED)) then
    ParseItem(item) /* lineare Struktur von item analysieren */
  if not(IsFlag(item, FLAG_PARSED_OK)) or
        LayerId(item, LAYERID_INVALID) then
    return /* Fehler! Verarbeitung abbrechen */
  if IsType(item, ITEM_IGNORE) then
    continue /* item überspringen */
  else if IsType(item, ITEM_TEXT) then
    for each layerStack in AnnotationLayerList do
      temp := layerStack.Peek()
      if IsType(temp, NODE_TEXT) then
        temp.AddItem(item)
      else
        layerStack.Push(new TextNode(item))
  else if IsType(item, ITEM_ELEMENT_START) then
    layerStack := GetStateStackFor(item)
    layerStack.Push(new ElementNode(item))
  else if IsType(item, ITEM_ELEMENT_EMPTY) then
    layerStack := GetStateStackFor(item)
    layerStack.Push(new EmptyElementNode(item))
  else if IsType(item, ITEN_ELEMENT_END) then
    layerStack := GetStateStackFor(item)
    p := 0
    while p < StackSize(layerStack) do
      temp := PeekAt(layerStack, p)
      if not(Flag(temp, FLAG_CLOSED)) then
        if Name(temp) ! = Name(i) then
          layerStack.MarkInvalid()
        break
      p := p + 1
    if p > 0 then
      while p > 0 do
        temp.InsertChild(layerStack.Pop(), 0)
        p := p - 1
      temp.SetFlag(FLAG_CLOSED)
for each layerStack in AnnotationLayerList do
  layer := AnnotationLayerFor(layerStack)
  if IsValid(layerStack) and (ItemCount(layerStack) == 1) then
    layer.SetRootNode(layerStack.Peek())
    layer.SetValidity(VALIDITY_WELLFORMED)
  else
    layer.SetRootNode(null)
    layer.SetValidity(VALIDITY_INVALID)
```

Abbildung 5.2: Der Pseudo-Code des Parsers

welchem Typ sie angehören, lösen sie unterschiedliche Aktionen im Parser aus. Sollte ein Fehler nur auf einer Annotationsebene auftreten, beispielsweise weil zu einem Element das End-Tag fehlt, wird die Verarbeitung auf ihr abgebrochen und ein Fehler angezeigt. Die anderen Annotationsebenen sind davon jedoch nicht betroffen.

Trifft der Parser auf ein Segment, bei dem nicht das Flag `FLAG_PARSED_OK` gesetzt ist oder die Layer-Id den Wert `LAYERID_INVALID` hat, wird der gesamte Vorgang abgebrochen und der Parser meldet einen Fehler.

Der Parser unterscheidet zwischen *offenen* und *geschlossenen* Knoten. Per Definition geschlossene Knoten sind TextNode- und EmptyElementNode-Knoten. Ein ElementNode-Knoten gilt als geschlossen, wenn das Segment für sein End-Tag verarbeitet wurde. Das Flag `FLAG_CLOSED` ist in dem Knoten-Objekt in diesem Fall gesetzt.

Bei der Verarbeitung eines Segments vom Typ `ITEM_ELEMENT_START` oder `ITEM_ELEMENT_EMPTY`, wird es auf den Stack der Annotationsebene, zu der es gehört, abgelegt. Als erstes wird dazu das Status-Objekt der Annotationsebene gesucht, das die passende Layer-Id besitzt und dann ein neues ElementNode- bzw EmptyElementNode-Objekt mit der Methode `Push` auf den Stack gelegt.

Wenn der Parser auf ein Segment vom Typ `ITEM_TEXT` trifft, wird es im allgemeinen Fall auf den Stack von *jeder* Annotationsebene abgelegt. Dazu wird ein neues TextNode-Objekt erzeugt und mit der `Push`-Operation auf den Stack geschoben. Wenn jedoch auf dem Stack einer Annotationsebene bereits ein TextNode-Objekt liegt, wird kein neues Objekt hinzugefügt. Das Segment wird mittels der Methode `AddItem` dem obenliegenden TextNode-Objekt hinzugefügt.

Sofern nicht auf allen Annotationsebenen bereits das Wurzelelement geöffnet wurde, wird überprüft, ob das Flag `FLAG_WHITESPACE_ONLY` gesetzt ist. Ist dies der Fall, wird das Text-Segment übergangen.

Wird ein Segment vom Typ `ITEM_ELEMENT_END` gelesen, versucht der Parser in der betreffenden Annotationsebene einen offenen ElementNode-Knoten zu finden, der zu dem End-Tag passt. Dazu verwendet der Parser die Methode `PeekAt` und sucht von der obersten Position des Stacks nach unten. Wird ein offener ElementNode-Knoten gefunden, der nicht zum gesuchten Knoten passt, gibt der Parser eine Fehlermeldung aus und bricht die Verarbeitung auf der Annotationsebene ab.

Sollte der passende Knoten gefunden werden, werden die Knoten, die im Stack über ihm liegen, entfernt und als Kind-Knoten dem Knoten zugeordnet. Danach wird das Flag `FLAG_CLOSED` gesetzt und der Knoten ist der oberste auf dem Stack.

Wenn alle Segmente verarbeitet worden sind, sollten die einzelnen Stacks der Annotationsebenen jeweils nur einen ElementNode-Knoten enthalten und nicht als invalide markiert sein. Ist dies der Fall wird der Knoten als Wur-

zelknoten für die entsprechende Annotationsebene gesetzt und der Status der Ebene auf `VALIDITY_WELLFORMED` gesetzt. In allen anderen Fällen wird die Ebene auf `VALIDITY_INVALID` gesetzt und der Wert `null` als Wurzelknoten gesetzt. Sollte eine "auto-created" Annotationsebene keine Knoten enthalten wird sie jetzt vom Parser gelöscht.

## 5.3 Mascarpone – der Editor

Das Programm "Mascarpone" ist eine Prototypen-Implementierung des Annotationswerkzeugs. Bislang sind noch nicht alle Funktionen implementiert, die einen vollwertigen Editor ausmachen, jedoch sind das Datenmodell und dessen Modifikation durch den Benutzer umgesetzt. Die Anwendung enthält eine Implementierung einer funktionsfähigen Version eines XMC-Parsers, der ein XMC-Dokument auf Wohlgeformtheit parsen kann. Weitere Funktionen umfassen das Laden und Speichern von XMC-Dokumenten und eine Reihe von Operationen, wie Cut, Copy, Paste, Undo und Redo, die für einen Editor im Allgemeinen üblich sind.

Das XMC-Dokument kann in der Anwendung interaktiv modifiziert werden und wenn der Benutzer einige Zeit[5] keine Änderungen vornimmt, wird durch den XMC-Parser die lineare Struktur überprüft und die hierarchische Struktur (siehe Abschnitt 5.2) aktualisiert.

Die einzelnen Annotationsebenen können einerseits manuell durch den Benutzer angelegt werden, andererseits erstellt der Parser automatisch eine neue Annotationsebene, wenn er auf eine unbekannte Annotationsebenen-Id trifft. Diese "auto-created" Annotationsebenen werden automatisch wieder entfernt, wenn sie keine Elemente mehr enthalten. In späteren Versionen des Editors soll der Benutzer die Möglichkeit haben, diese Annotationsebenen permanent zu machen, d. h. sie werden den manuell erstellten gleichgestellt und nicht mehr entfernt, wenn sie leer sind.

Der Prototyp implementiert eine rudimentäre Plugin-Schnittstelle und Verwaltung. Diese soll genutzt werden, um sowohl Import- und Export-Filter als auch verschiedene Implementierungen der Schemasprachen für Annotationsschemata (dynamisch) laden zu können.

Abbildung 5.3 zeigt ein Bildschirmfoto vom Hauptfenster der Anwendung. Der Hauptteil des Fensters wird durch eine grosse Texteingabekomponente mit Zeilennummerierung eingenommen. In ihr kann das XMC-Dokument eingegeben und modifiziert werden. Auf der Abbildung ist das Dokument aus Abbildung 3.1 geladen.

Links von der Eingabekomponente ist ein Bereich abgegrenzt, in dessen oberen Teil eine Liste der vorhandenen Annotationsebenen angezeigt wird. Jede Annotationsebene wird durch ihre Annotationsebenen-Id dargestellt. In

[5] In der Prototypen-Implementierung 2 Sekunden.

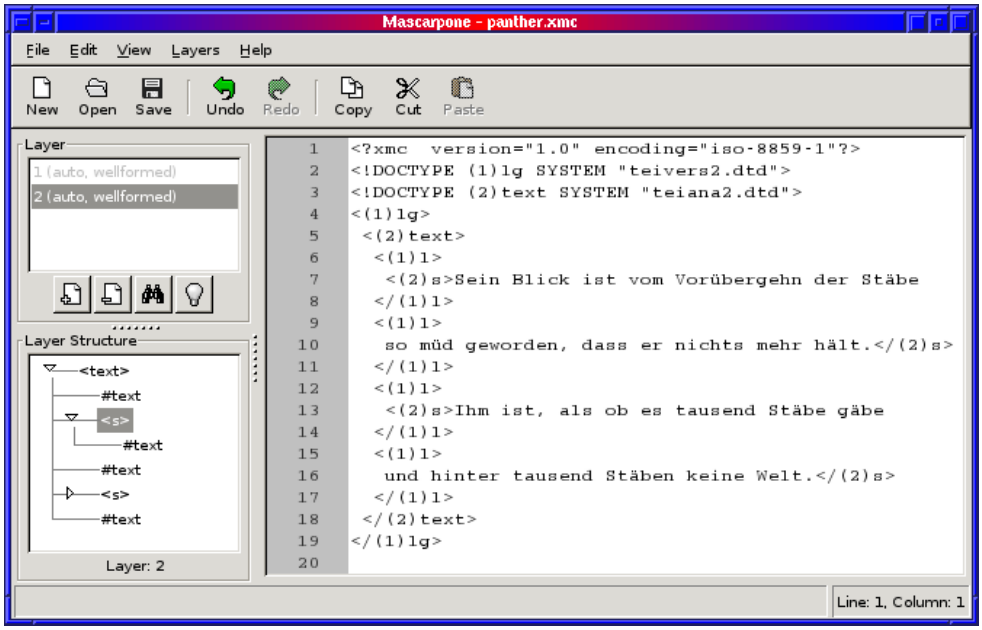


Abbildung 5.3: Ein Screenshot des Editors (1)

den Klammern werden weitere Informationen zu einer Ebene dargestellt. Das Bildschirmfoto zeigt zwei Annotationsebenen. Die graue Darstellung und das Wort “auto” in den Klammern zeigen an, dass es sich um “auto-created “ Annotationsebene handelt. Eine vom Benutzer erstellte Ebene würde schwarz und ohne das Wort “auto” angezeigt werden. Beide Annotationsebenen sind wohlgeformt. Dies wird durch das Wort “wellformed” in den Klammen angezeigt. Wenn die Ebene valide ist, wird das Wort “valid” angeben. Eine Ebene, die weder valide nocht wohlgeformt ist, wird als “invalid” angezeigt.

Unterhalb der Liste der Annotationsebenen sind vier Schaltflächen angeordnet. Mit ihnen können Annotationsebenen hinzugefügt und entfernt werden. Es ist bislang nur das Hinzufügen einer Annotationsebene implementiert. Wenn eine Annotationsebene in der Liste markiert ist, kann durch Drücken der Schaltfläche mit dem Fernglas eine Strukturansicht der Ebene angezigt werden. Die Schaltfläche mit der Glühbirne ist zur Zeit ohne Funktion, aber soll in einer späteren Version detaillierte Informationen zu einer liefern.

Unterhalb der Liste der Annotationsebenen kann die Struktur einer Annotationsebene als Baumansicht angezeigt werden. Die einzelnen Äste der Baumansicht lassen sich unabhängig voneinander auf- oder zuklappen. Das Bildschirmfoto zeigt die Ansicht der Annotationsebene “2”. In einer überarbeiteten Version der Anwendung kann der Anwender durch Anklicken eines Knotens den Cursor in der Eingabekomponente auf das entsprechende Element positionieren. Bei einem Text-Knoten würde der Cursor auf den Beginn des Segments

gesetzt, bei Element-Knoten beim ersten Klick auf das Start-Tag und bei einem weitern Klick auf das End-Tag. Bei einem Empty-Tag gibt es natürlich keine Unterscheidung zwischen Start- und End-Tag. Ein Doppelklick auf einen Knoten könnte in der Eingabekomponente die Region selektieren, die er überspannt.

Im oberen Teil des Fensters befindet sich eine Werkzeugleiste, die auch als Toolbar bezeichnet wird. Sie ermöglicht den schnellen Zugriff auf die wichtigsten Funktionen der Anwendung. Darüber befindet sich die obligatorische Menüleiste, die die üblichen Funktionen bietet.

Am unteren Rand des Anwendungsfenster ist eine Statuszeile. Die Applikation kann dort Statusmeldungen ausgeben. Im rechten Teil der Statusleiste wird angezeigt, in welcher Zeile und Spalte sich der Cursor in der Eingabekomponente befindet.

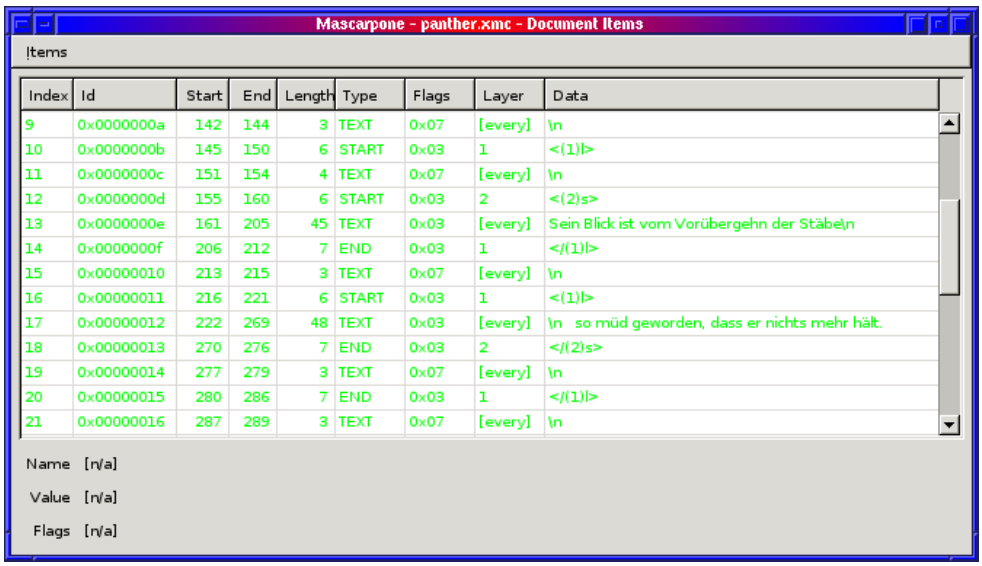


Abbildung 5.4: Ein Screenshot des Editors (2)

Die Applikation bietet dem Benutzer die Möglichkeit die lineare Struktur zu inspezieren. Die Abbildung 5.4 zeigt den DocumentItem-Inspektor. Das Fenster kann durch das Drücken der Tastenkombination CTRL-1 oder durch die Auswahl des Eintrags “Show Linear Structure” aus dem Menü “View” geöffnet werden.

Der Inspektor zeigt eine Liste der DokumentItem, und damit der Segmente, die ein Dokument enthält, an. Jede Zeile ist in mehrere Spalten unterteilt, die verscheidene Informationen darstellen.

Die erste Spalte gibt die Position des Segments in der Liste der Segmente des Dokuments an. In der nächsten Spalte ist die eindeutige Id angezeigt, die

jedem Segment zugeordnet ist.

Die Spalten "Start" und "End" geben die Start- und End-Position des Segments an. Diese Werte sind in Byte angeben. In der Spalte "Length" kann man die Länge des Segments anlesen. Auch dieser Wert ist in Byte angegeben.

Der Typ eines Segments wird in der Spalte "Typ" angezeigt. Der Wert "IGNORE" bedeutet, dass das Segment vom Parser ignoriert wird. Der Wert "TEXT" steht für ein Text-Segment, "START" für ein Start-Tag, "END" für ein End-Tag und "EMPTY" für ein Empty-Tag.

Die Flags, die einem Segment zugeordnet sein, werden in der Spalte "Flags" angeben. Um Platz zu sparen, werden sie hier in Hexadezimalzahlen angezeigt. Die einzelnen Flags sind als verschiedene Bits einer Integerzahl implementiert. Die Zahl `0x0` bedeutet, dass kein Flag gesetzt ist; die Zahl `0x1` bedeutet, dass das Flag `FLAG_PARSED` gesetzt ist. Wenn die Flags `FLAG_PARSED` und `FLAG_WHIESPACE_ONLY` gesetzt sind, ist die Zahl `0x5`.

Die Spalte "Layer" zeigt an, auf welcher Annotationsebene sich das Segment befindet. Wenn ein Segment nicht auf allen Annotationsebenen vorhanden ist, wie beispielsweise ein Text-Segment, wird hier der Wert "[every]" angegeben. Ansonsten wird die Annotationsebenen-Id angezeigt.

In der Spalte "Data" werden die Dokumentdaten angezeigt, die ein Segment bilden.

Die Modifikation des Datenmodells wird durch die Implementierung des Editors vorgenommen. Die Änderungen des Benutzers, also das Löschen oder Einfügen von Zeichen oder das Verwenden der "Cut-and-Paste"-Funktionen, werden durch die Eingabekomponente in Einfüge- und Lösch-Ereignisse umgesetzt. Diese Ereignisse werden von den Methoden `InsertText` und `DeleteText` verarbeitet und modifizieren das Datenmodell. Sie bekommen jeweils eine Position und eine Länge als Argument übergeben. Bei einer Einfügeoperation geben die beiden Werte die Anzahl der Zeichen in Byte an, die an einer bestimmten Position eingefügt wurden. Analog dazu geben sie bei einer Löschoperation die Anzahl der Zeichen in Byte an, die ab einer bestimmten Position gelöscht wurden.

In beiden Fällen wird die Zeichenkette als erstes in Teilstücke zerlegt sofern sie einen Tag-Begrenzer ("`<`" oder "`>`") enthält. Die Teilstücke bestehen nun entweder aus "reinem Text" (auch mehrere Zeichen) oder genau einem Tag-Begrenzer. Zu jedem Teilstück wird nun mit Hilfe einer binären Suche das Segment gesucht, in das es eingefügt oder gelöscht werden muss. Dies wird als Segment $i$ bezeichnet.

Als erstes wird das Einfügen eines Teilstücks in ein Segment betrachtet. Je nachdem, welches Zeichen sich an der Position befindet, an der das Teilstück eingefügt werden soll, müssen unterschiedliche Aktionen durchgeführt werden. Die Spalte "Segment" gibt an, welches Zeichen an der Einfügeposition $p$ im Segment steht und die Spalte "Teilstück", welche Zeichen sich im Teilstück

befinden.

| Segment | Teilstück | Aktion |
|---|---|---|
| Text | Text | Teilstücks in Segment $i$ einfügen |
| < | Text oder > | Wenn das letzte Zeichen von Segment $i-1$ kein Tag-Begrenzer ist, dann das Teilstück an Segment $i-1$ anhängen. Andernfalls das Teilstück als neues Segment vor Segment $i$ einfügen. |
| > | Text oder < | Wenn das erste Zeichen von Segment $i+1$ kein Tag-Begrenzer ist, dann das Teilstück dem Segment $i+1$ voranstellen. Andernfalls das Teilstück als neues Segment nach Segment $i$ einfügen. |
| Text | < | Segment $i$ an Position $p$ teilen und ein neues Segment $j$ einfügen. Das Teilstück gehört zu $j$. |
| Text | > | Segment $i$ an Position $p$ teilen und ein neues Segment $j$ einfügen. Das Teilstück gehört zu $i$. |
| < | < | Teilstück als ein neues Segment vor $i$ einfügen. |
| > | > | Teilstück als ein neues Segment nach $i$ einfügen. |

Natürlich können die Zeichen des vorhergehenden bzw. nachfolgenden Segments nur getestet werden, wenn die Segmente überhaupt existieren. Sollte dies nicht der Fall ist, wird ein neues Segment an der entsprechenden Stelle eingefügt.

Beim Löschen wird das Teilstück aus dem entsprechenden Segment entfernt. Wenn am Anfang oder am Ende eines Segments gelöscht wurde, muss überprüft werden, ob das Segment mit dem vorherigen beziehungsweise nachfolgenden vereinigt werden kann. Segmente können immer vereinigt werden, wenn zwei Text-Teile, also keine Tag-Begrenzer, aufeinander stossen. Nach dem Löschen vorhandene Elemente der Länge 0 werden aus der linearen Struktur entfernt.

Bei einer Überarbeitung der Prototypen-Implementierung sollten die Programmteile, die die lineare Struktur modifizieren, in die Klassen des Datenmodells integriert werden. Dies würde zu einem konsistenteren Datenmodell führen. Diese Funktionen wurden im Editor implementiert, weil anfangs ein höherer Grad der Abstraktion zwischen Datenmodell und Editor angestrebt wurde.

# Kapitel 6

# Offene Fragen und Ausblick

## 6.1 Offene Fragen

Im Rahmen der Bearbeitung der Aufgabenstellung haben sich einige Fragen gestellt, die bislang unbeantwortet geblieben sind.

Bei der aktuellen Implementierung wird das Datenmodell durch Änderungen des XMC-Dokuments im Editor modifiziert. Jedoch wäre es auch denkbar, das Datenmodell durch Modifikationen an den Knoten im hierarchischen Modell zu ändern. Leider sind die herangehensweisen sehr unterschiedlich. Bei den "eingabegetriebenen" Änderungen im Editor wird die lineare Struktur modifiziert und die hierarchische Struktur durch den Parser aufgebaut. Bei einer Änderung in der Knoten-Struktur ergeben sich eine Reihe von Problemen.

Als erstes wird die Änderung nur auf einer Annotationsebene durchgeführt. Wie verhält sie sich zu den anderen Ebenen? Das Löschen eines Element-Knotens ist noch relativ einfach zu realisieren, da nur die Segmente, die das Start- und End-Tag (oder das Empty-Tag) abbilden, entfernt werden müssten. Was soll aber mit eventuell eingeschlossenen Text-Knoten passieren? Sollen sie auch gelöscht werden? Wenn ja, müssen sie natürlich auch aus allen anderen Ebenen entfernt werden.

Das Einfügen eines Knotens bereitet ebenso Schwierigkeiten. An welcher Stelle sollen die Segmente für das Start- und End-Tag in die lineare Struktur eingesetzt werden? Wie verhalten sich die anderen Ebenen, wenn ein Text-Segment in einer Annotationsebene eingesetzt wird? An welcher Position soll es in die lineare Struktur eingesetzt werden? Muss es eventuell "gesliced"[1] werden?

Eine weitere Frage, die sich stellt, ist die nach einer effizienten Aktualisierung der hierachischen Struktur des Datenmodells. In der aktuellen Implementierung werden beim Erstellen der hierarchischen Struktur alle Knoten-Hierarchien gelöscht und komplett neu aufgebaut. In wieweit bietet sich die

[1] siehe Abschnitt 4.2.2

Möglichkeit, nur Teile der Hierarchie zu löschen und neu aufzubauen, wenn nur geringe Modifikationen am Datenmodell vorgenommen wurden?

Ein erster Schritt könnte die Klassifizierung der Modifikationen in "kontext-frei" und nicht "kontext-frei" sein. Eine Veränderung an einem bestehenden Text-Segment durch Hinzufügen oder Löschen von Zeichen ist eine "kontext-freie" Modifikation, da sich die hierarchische Struktur nicht ändert. Wenn jedoch ein ganzes Element oder Text-Segment gelöscht wird, ändert sich auch die hierarchische Struktur. Eventuell lässt sich durch inkrementelles Parsen ein Teil der bestehenden Struktur wiederverwenden. So müsste sie nur an den betroffenen Stellen modifiziert werden.

Weiterhin kann es mit grossen Dokumenten[2] ein Performanzproblem beim Aktualisieren der linearen Struktur geben. Da die Änderungen am Datenmodell synchron zu den Benutzereingaben vorgenommen werden, bemerkt der Benutzer bei grossen Dokumenten teilweise Verzögerungen bei der Aktualisierung der Graphischen Benutzeroberfläche und der Verarbeitung der Tastartureingaben.

Die Liste, in der die Segmente gespeichert werden, wird mit Hilfe einer binären Suche durchsucht. Eventuell könnten die Vergleiche, die benötigt werden, durch eine zweistufige binäre Suche veringert werden. Dazu wird die Liste in $n$ gleichgrosse Teile, sogenannte "Buckets", zerlegt. Ein Bucket speichert die kleinste Start-Position und die grösste End-Positon, die die in ihm enthaltenen Segmente haben. Im ersten Schritt könnte man den "Bucket" suchen, in dem sich die gesuchte Position befindet und dann innerhalb dieses "Buckets" nach dem entsprechenden Segment suchen.

Die Aktualisierungen der linearen Struktur könnten auch zeitversetzt erfolgen. Erst wenn der Benutzer mit dem Curser die Zeile wechselt oder eine Tag-Begrenzung überschreitet wird die Struktur aktualisiert.

## 6.2 Ausblick

Mit XML CONCOR wird eine einfache, aber dennoch mächtige Dokumentsyntax zur Annotation von multihierarchisch strukturierten Dokumenten definiert. XMC soll keine neue Markup-Sprache definieren, sondern als Zwischenformat bei der Bearbeitung dieser Annotationen helfen. Durch die Anlehnung der Syntax an XML sollte es jedem, der bereits Erfahrungen mit XML hat, leicht fallen, Dokumente mit XMC zu annotieren. Durch die zwei Verarbeitungsmodelle kann ein Anwender sich das Modell aussuchen, das seiner Problemstellung und deren Lösung am nächsten kommt.

Der vorliegende Prototyp des Annotationswerkzeugs stellt eine Implementierung des Datenmodells und eines Parsers zur Verfügung, jedoch fehlen noch

[2] Je nach Ausstattung des Rechners Dokumente, die mehr als 100000 Segmente in der linearen Struktur enthalten.

viele Eigenschaften, die einen vollwertigen Editor ausmachen.

Der Editor solle in der Lage sein, das Dokument zu validieren und eventuelle Fehler, beipsielsweise durch farbliche Hervorhebung in der Texteingabekomponente, anzuzeigen. Außerdem sollte der Benutzer bei der Annotation unterstützt werden. Dem Benutzer könnte beispielsweise, wenn er eine bestimmte Tastenkombination drückt, einen Dialog mit einer Liste von Elementen präsentiert werden, die eine Dokument-Grammatik an der aktuellen Cursorposition erlauben würde. Element-Namen könnten auch automatisch bei der Eingabe komplettiert werde.

Dazu muss das Programm in der Lage, sein verschiedene Dokument-Grammatiken, wie DTD, XML-Schema oder RelaxNG, zu verarbeiten. Für diese Grammatiken müsste jeweils ein Parser implementiert werden oder eine bestehende Implementation in das Programm eingebunden werden. Weiterhin müss der XMC-Parser erweitert werden, so dass er ein XMC-Dokument auch validieren kann.

Ein weitere wichtige Eigenschaft ist das Syntax-Highlighting. Der Editor könnte verschiedene Elemente eines XMC-Dokuments in verschiedenen Farben oder Schriftattributen darstellen, um die Lesbarkeit zu erhöhen. So könnten die Annotationsebenen-Id, die Element-Namen oder unterschiedliche Regionen im Dokument wie eine Annotationsschema-Deklaration oder ein Kommentar sich farblich voneinander abheben. Alternativ könnte für jede Annotationsebene eine Farbe gewählt werden und alle Tags werden dann in der Farbe "ihrer" Annotationsebene dargestellt.

Weiterhin muss der Editor in der Lage sein, verschiedene Formate zu importieren und exportieren. Besonders wichtig ist das Einlesen von $n$ primärdatenidentischen XML-Dokumenten. Beim Laden muss eine Whitespace-Normalisierung durchgeführt werden und die XML-Dateien können dann zu einem XMC-Dokument zusammengeführt werden. Die Whitespace-Normalisierung könnte auch von einem externen Programm durchgeführt werden, jedoch wäre es für den Benutzer angenehmer, wenn der Editor diese Aufgabe übernommen könnte.

Natürlich sollte ein XMC-Dokument auch wieder in $n$ XML-Dokumente gespeichert werden können. So kann der Benutzer das Dokument bequem annotieren und nach dem Exportieren mit anderer Software weiterverarbeiten. Der Import und Export von verschiedenen anderen Formaten wie beispielsweise TexMECS oder LMNL sollte auch möglich sein.

Bei der Weiterentwicklung des Editors müsste das Datenmodell des Verarbeitungsmodells überarbeitet werden. Einerseits sollten die Klassen konsistente und eindeutige Namen, wie XMCDocument, XMCElement oder XMCAttribute bekommen und anderseits könnte der Klassenstruktur neu organisiert werden. Die Klassen `EmptyElementNode` und `ElementNodeBase` sollten wegfallen, da eine eigene Klasse für leere Elemente in der hierarchischen Struktur eigentlich unnötig ist.

Als “Proof-Of-Concept” zeigt die Prototypen-Implementierung des Editors und des Datenmodells, dass das Modell realisierbar ist und bietet einen guten Startpunkt für eine vollwertige Implementierung.

# Literaturverzeichnis

[1] 8879:1986, ISO: *Text and office systems – Standard Generalized Markup Language (SGML).* International Organization for Standardization, Geneva, 1986.

[2] Bray, Tim, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau und John Cowan: *Extensible Markup Language (XML) 1.1.* World Wide Web Consortium, 2004.

[3] Conrady, Karl Otto (Herausgeber): *Der Neue Conrady. Das große deutsche Gedichtsbuch*, Seite 561. Patmos Verlag, 2000.

[4] Czmiel, Alexander: *XML for Overlapping Structures (XfOS) using a non XML Data Model.* In: *The Joint International Conference: ALLC/ACH 2004*, Gothenburg, Sweden, 2004.

[5] DeRose, Steven: *Markup Overview: A Review and a Horse.* In: *Extreme Markup Languages Proceedings*, Montreal, 2004.

[6] Durusau, Patrick und Matthew Brook O'Donnell: *Coming down from the trees: Next step in the evolution of markup?* In: *Extreme Markup Languages Proceedings (late breaking paper)*, Montreal, 2002.

[7] Durusau, Patrick und Matthew Brook O'Donnell: *Concurrent Markup of XML Documents.* In: *XML Conference & Exposition 2002*, Barcelona, 2002.

[8] Fallside, David C. und Priscilla Walmsley: *XML Schema Part 0: Primer Second Edition.* World Wide Web Consortium, 2004.

[9] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1997.

[10] Goldfarb, Charles F.: *The SGML Handbook.* Clarendon Press, Oxford, 1990.

[11] Hilbert, Mirco: *MuLaX – ein Modell zur Verarbeitung mehrfach XML-strukturierter Daten.* Diplomarbeit, Universtität Bielefeld, Bielefeld, 2005.

[12] Hilbert, Mirco, Oliver Schonefeld und Andreas Witt: *Making CONCUR work.* In: *Extreme Markup Languages Proceedings*, Montreal, 2005.

[13] Hors, Arnaud Le, Philippe Le Hégaret, Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion und Steve Byrne: *Document Object Model (DOM) Level 3 Core Specification*. World Wide Web Consortium, 2004.

[14] Nicol, Gavin: *Core Range Algebra: Towards a Formal Model of Markup*. In: *Extreme Markup Languages Proceedings*, Montreal, 2002.

[15] Smart, Julian, Kevin Hock und Stefan Csomor: *Cross-Platform GUI Programming with wxWidgets*. Prentice Hall, 2005.

[16] Sperberg-McQueen, C. M. and Lou Burnard (editors): *Guidelines for Text Encoding and Interchange*. published for the TEI Consortium by Humanities Computing Unit, University of Oxford, 2004.

[17] Sperberg-McQueen, C. M. und Claus Huitfeldt: *Concurrent Document Hierarchies in MECS and SGML*. In: *The Joint International Conference: ALLC/ACH 1998*, Debrecen, Hungary, 1998.

[18] Stroustrup, Bjarne: *The C++ Programming Language*. Addison-Wesley, 1997.

[19] Tennison, Jeni und Wendell Piez: *The Layered Markup and Annotation Language (LMNL)*. In: *Extreme Markup Languages Proceedings (late breaking paper)*, Montreal, 2002.

# Index

# Quellcode

Der folgende Abschnitt enthält die C++-Header-Dateien, der Klassen, die das Datenmodell implementieren. Die komplette Implementierung der Klassen wurde aus Platzgründen nicht mit hinzugefügt. Der komplette Quelltext ist jedoch auf der beiliegenden CD-ROM zu finden.

Document.hpp

```
1  /* $id$
2   * Copyright (C) 2005 Oliver Schonefeld
3   * All rights reserved
4   */
5  #ifndef DOCUMENT_HH
6  #define DOCUMENT_HH
7  #include <sys/types.h>
8  #include <algorithm>
9  #include <functional>
10 #include <vector>
11 #include <map>
12 #include <wx/string.h>
13 #include "DataAccess.hpp"
14 #include "Attribute.hpp"
15
16
17 /*****************************************************************************
18  * general definitions
19  *****************************************************************************/
20
21 typedef signed int layerid_t;
22 const layerid_t LAYERID_EVERY_LAYER =  0;
23 const layerid_t LAYERID_INVALID   = -1;
24 typedef signed char layergeneration_t;
25
26 typedef unsigned int itemid_t;
27 const itemid_t ITEMID_INVALID = 0;
28
29 // forward declaration
30 class MultiLayerDocument;
31 class MultiLayerDocumentParser;
32 class DocumentNode;
33
34
35 /*****************************************************************************
36  * declaration class AnnotationLayer
37  *****************************************************************************/
38
39 class AnnotationLayer {
40   friend class MultiLayerDocument;
41   friend class MultiLayerDocumentParser;
42 public:
43   typedef enum {
44     VALIDITY_UNKNOWN = 0,
45     VALIDITY_INVALID,
46     VALIDITY_WELLFORMED,
47     VALIDITY_VALID,
48   } Validity;
49
50   virtual ~AnnotationLayer();
51
52   layerid_t GetId() const;
53
54   const wxString& GetPrefix() const;
55
56   void SetPrefix(const wxString& prefix);
57
58   DocumentNode* GetRootNode() const;
59
60   bool IsAutoCreated() const;
61
62   Validity GetValidity() const;
63
64   layergeneration_t GetParsedGeneration() const;
65
66 private:
67   MultiLayerDocument *m_document;
68   layerid_t m_layerid;
69   wxString m_prefix;
70   DocumentNode *m_root;
71   Validity m_validity;
```

```
72      size_t m_parsedcount;
73      layergeneration_t m_generation;
74
75      explicit AnnotationLayer(MultiLayerDocument *document,
76                               const wxString& prefix, const bool auto_allocated);
77
78      void SetRootNode(DocumentNode *root);
79
80      size_t GetParsedItemCount() const;
81
82      void SetParsedItemCount(const size_t count);
83
84      void SetValidity(const Validity validity);
85
86    }; // class AnnotationLayer
87
88
89    inline layerid_t AnnotationLayer::GetId() const {
90      return (m_layerid);
91    }
92
93
94    inline const wxString& AnnotationLayer::GetPrefix() const {
95      return (m_prefix);
96    }
97
98
99    inline DocumentNode* AnnotationLayer::GetRootNode() const {
100     return (m_root);
101   }
102
103
104   inline size_t AnnotationLayer::GetParsedItemCount() const {
105     return (m_parsedcount);
106   }
107
108
109   inline bool AnnotationLayer::IsAutoCreated() const {
110     return (m_layerid < LAYERID_INVALID);
111   }
112
113
114   inline AnnotationLayer::Validity AnnotationLayer::GetValidity() const {
115     return (m_validity);
116   }
117
118
119   inline layergeneration_t AnnotationLayer::GetParsedGeneration() const {
120     return (m_generation);
121   }
122
123
124   /*****************************************************************************
125    * declaration abstract class DocumentItem
126    *****************************************************************************/
127
128   class DocumentItem {
129     friend class MultiLayerDocument;
130     friend class MultiLayerDocumentParser;
131   public:
132     typedef enum {
133       ITEM_TEXT,
134       ITEM_ELEMENT_START,
135       ITEM_ELEMENT_END,
136       ITEM_ELEMENT_EMPTY,
137       ITEM_IGNORE
138     } ItemType;
139
140     // remember to check the SetFlags method when adding new flags!
141     enum {
142       FLAG_PARSED            = 1,
143       FLAG_PARSED_OK         = 2,
144       FLAG_WHITESPACE_ONLY   = 4,
145     } Flags;
146
147     virtual ~DocumentItem();
148
149     const itemid_t GetId() const;
150
151     layerid_t GetLayerId() const;
152
153     wxString GetData() const;
154
155     bool IsInside(const int offset) const;
```

```
156
157       bool IsEmpty() const;
158
159       int GetOffset() const;
160
161       int GetEndOffset() const;
162
163       int GetLength() const;
164
165       void SetLength(const int length);
166
167       void Move(const int shift);
168
169       void IncLength(const int length);
170
171       void DecLength(const int length);
172
173       bool AreOffsetsValid() const;
174
175       ItemType GetType() const;
176
177       wxString GetName() const;
178
179       wxString GetValue() const;
180
181       int GetFlags() const;
182
183     protected:
184       MultiLayerDocument *m_document;
185       const itemid_t m_id;
186       ItemType m_type;
187       layerid_t m_layerid;
188       int m_offset;
189       int m_length;
190       unsigned char m_flags;
191       wxString m_name;
192       wxString m_value;
193       explicit DocumentItem(MultiLayerDocument *document,
194                             const ItemType itype,
195                             const int offset,
196                             const int length);
197
198     private:
199       void SetName(const wxString& name);
200
201       void SetValue(const wxString& value);
202
203       void SetLayerId(const layerid_t layerid);
204
205       void SetType(const ItemType itype);
206
207       void SetFlags(const int flags);
208
209     }; // abstract class DocumentItem
210
211
212     inline const itemid_t DocumentItem::GetId() const {
213       return (m_id);
214     }
215
216
217     inline layerid_t DocumentItem::GetLayerId() const {
218       return (m_layerid);
219     }
220
221
222     inline bool DocumentItem::IsEmpty() const {
223         return (m_length == 0);
224     }
225
226
227     inline bool DocumentItem::IsInside(const int offset) const {
228         return ((m_offset <= offset) && (offset <= (m_offset + m_length - 1)));
229     }
230
231
232     inline int DocumentItem::GetOffset() const {
233       return (m_offset);
234     }
235
236
237     inline int DocumentItem::GetEndOffset() const {
238       assert(m_length > 0);
239       return (m_offset + m_length - 1);
```

```
240   }
241
242
243   inline int DocumentItem::GetLength() const {
244     return (m_length);
245   }
246
247
248   inline DocumentItem::ItemType DocumentItem::GetType() const {
249     return (m_type);
250   }
251
252
253   inline int DocumentItem::GetFlags() const {
254     return (m_flags);
255   }
256
257
258   inline void DocumentItem::SetLayerId(const layerid_t layerid) {
259     m_layerid = layerid;
260   }
261
262
263   /*****************************************************************************
264    * declaration class DocumentNode
265    *****************************************************************************/
266
267   class DocumentNode {
268   public:
269     typedef enum {
270       NODE_TEXT,
271       NODE_ELEMENT,
272       NODE_ELEMENT_EMPTY
273     } NodeType;
274
275     // remember to check the SetFlags method when adding new flags!
276     typedef enum {
277       FLAG_CLOSED = 1
278     } node_flags_t;
279
280     virtual ~DocumentNode();
281
282     virtual const itemid_t GetId() const = 0;
283
284     virtual NodeType GetType() const = 0;
285
286     virtual wxString GetName() const = 0;
287
288     virtual wxString GetValue() const = 0;
289
290     DocumentNode* GetParent() const;
291
292     virtual size_t GetChildrenCount() const = 0;
293
294     virtual DocumentNode* GetChild(const size_t idx) = 0;
295
296     virtual void AppendChild(DocumentNode *node) = 0;
297
298     virtual void InsertChild(DocumentNode *node, size_t idx) = 0;
299
300     int GetFlags() const;
301
302     void SetFlags(const int flags);
303
304     void SetParent(DocumentNode *parent);
305
306   protected:
307     DocumentNode* m_parent;
308     unsigned char m_flags;
309
310     explicit DocumentNode();
311
312   }; // class DocumentNode
313
314
315   inline DocumentNode* DocumentNode::GetParent() const {
316     return (m_parent);
317   }
318
319
320   inline int DocumentNode::GetFlags() const {
321     return (m_flags);
322   }
323
```

```
324  /*****************************************************************************
325   * declaration class MultiLayerDocumentListener
326   *****************************************************************************/
327
328  class MultiLayerDocumentListener {
329  public:
330    virtual void DocumentChanged(MultiLayerDocument *document) = 0;
331
332  }; // class MutliLayerDocumentListener
333
334
335  /*****************************************************************************
336   * declaration class MultiLayerDocument
337   *****************************************************************************/
338
339  class MultiLayerDocument {
340    friend class AnnotationLayer;
341    friend class DocumentItem;
342    friend class MultiLayerDocumentParser;
343  public:
344    explicit MultiLayerDocument(DataAccess *dataaccess);
345
346    virtual ~MultiLayerDocument();
347
348    AnnotationLayer* GetLayer(const layerid_t layerid) const;
349
350    AnnotationLayer* GetLayer(const wxString& prefix) const;
351
352    AnnotationLayer* GetLayerByIndex(const size_t idx) const;
353
354    AnnotationLayer* CreateLayer(const wxString& prefix);
355
356    void DeleteLayer(const wxString& prefix);
357
358    size_t GetLayerCount() const;
359
360    DocumentItem* GetItem(const size_t idx) const;
361
362    DocumentItem* GetLastItem() const;
363
364    DocumentItem* Insert(const size_t idx, const int offset, const int length);
365
366    void Delete(const size_t idx);
367
368    size_t GetCount() const;
369
370    void RevalidateItemType(const size_t idx);
371
372    bool IsEmpty() const;
373
374    void AddDocumentListener(MultiLayerDocumentListener *listener);
375
376    void RemoveDocumentListener(MultiLayerDocumentListener *listener);
377
378    size_t GetDocumentListenerCount() const;
379
380    void Freeze();
381
382    void Thaw();
383
384    //  void BuildStructure();
385
386  private:
387    typedef std::vector<AnnotationLayer*> AnnotationLayerVector;
388    typedef std::vector<DocumentItem*> DocumentItemVector;
389    typedef std::vector<MultiLayerDocumentListener*> DocumentListenerVector;
390    DataAccess *m_dataaccess;
391    bool m_changed;
392    bool m_frozen;
393    AnnotationLayerVector m_layers;
394    DocumentItemVector m_items;
395    DocumentListenerVector m_listeners;
396
397    DataAccess* GetDataAccess() const;
398
399    AnnotationLayer* CreateAutoAllocatedLayer(const wxString& prefix);
400
401    void DeleteLayerInternal(const layerid_t layerid);
402
403    void NotifyDocumentListeners();
404
405    DocumentItem::ItemType ClassifyStringData(const wxString& data) const;
406
407  }; // class MultiLayerDocument
```

```
408
409
410 inline size_t MultiLayerDocument::GetLayerCount() const {
411   return ((size_t) m_layers.size());
412 }
413
414
415 inline size_t MultiLayerDocument::GetCount() const {
416   return ((size_t) m_items.size());
417 }
418
419
420 inline bool MultiLayerDocument::IsEmpty() const {
421   return (m_items.empty());
422 }
423
424
425 inline size_t MultiLayerDocument::GetDocumentListenerCount() const {
426   return ((size_t) m_listeners.size());
427 }
428
429
430 inline DataAccess* MultiLayerDocument::GetDataAccess() const {
431   return (m_dataaccess);
432 }
433
434 #endif /* DOCUMENT_HH */
```

MultiLayerDocumentParser.hpp

```
1  /* $id$
2   * Copyright (C) 2005 Oliver Schonefeld
3   * All rights reserved
4   */
5  #ifndef MULTILAYERDOCUMENTPARSER_HH
6  #define MULTILAYERDOCUMENTPARSER_HH
7  #include <map>
8  #include <vector>
9  #include "Document.hpp"
10
11 /*****************************************************************************
12  * declaration class MultiLayerDocumentParser
13  *****************************************************************************/
14
15 class MultiLayerDocumentParser {
16 public:
17   explicit MultiLayerDocumentParser();
18
19   ~MultiLayerDocumentParser();
20
21   void ParseDocument(MultiLayerDocument *document);
22
23   void Reset();
24
25 private:
26   class LayerStack : private std::vector<DocumentNode*> {
27   public:
28
29     explicit LayerStack(AnnotationLayer *layer, const size_t size = 0);
30
31     virtual ~LayerStack();
32
33     layerid_t GetLayerId() const;
34
35     AnnotationLayer* GetLayer() const;
36
37     DocumentNode* GetRoot() const;
38
39     void Push(DocumentNode* node);
40
41     DocumentNode* Pop();
42
43     DocumentNode* Peek() const;
44
45     DocumentNode* PeekAt(const size_t back_idx) const;
46
47     bool IsEmpty() const;
48
49     size_t GetSize() const;
50
51     size_t GetElementCount() const;
52
53     size_t GetTotalCount() const;
54
55     bool IsOutsideOfRoot() const;
```

```
56
57      bool IsLayerValid() const;
58
59      void MarkInvalid();
60
61      void DumpStack() const;
62
63    private:
64      AnnotationLayer *m_layer;
65      DocumentNode *m_root;
66      size_t m_elementcount;
67      size_t m_totalcount;
68      bool m_invalid;
69    }; // class LayerStack
70
71    typedef std::map<layerid_t, LayerStack*> stackmap_t;
72    MultiLayerDocument *m_document;
73    stackmap_t m_stacks;
74
75    bool OneOutsideRoot() const;
76
77    bool BuildStructure(DocumentItem *item);
78
79    void ParseItem(DocumentItem *item);
80
81  }; // class MultiLayerDocumentParser
82
83
84  inline layerid_t MultiLayerDocumentParser::LayerStack::GetLayerId() const {
85    return (m_layer->GetId());
86  }
87
88
89  inline AnnotationLayer*
90  MultiLayerDocumentParser::LayerStack::GetLayer() const {
91    return (m_layer);
92  }
93
94
95  inline bool MultiLayerDocumentParser::LayerStack::IsEmpty() const {
96    return (empty());
97  }
98
99
100 inline size_t MultiLayerDocumentParser::LayerStack::GetSize() const {
101   return (size());
102 }
103
104
105 inline size_t MultiLayerDocumentParser::LayerStack::GetElementCount() const {
106   return (m_elementcount);
107 }
108
109
110 inline size_t MultiLayerDocumentParser::LayerStack::GetTotalCount() const {
111   return (m_totalcount);
112 }
113
114
115 inline bool MultiLayerDocumentParser::LayerStack::IsLayerValid() const {
116   return (m_invalid == false);
117 }
118
119 #endif  /* MULTILAYERDOCUMENTPARSER_HH */
```

TextNode.hpp

```
1   /* $id$
2    * Copyright (C) 2005 Oliver Schonefeld
3    * All rights reserved
4    */
5   #ifndef TEXTNODE_HH
6   #define TEXTNODE_HH
7   #include <vector>
8   #include "Document.hpp"
9
10  /**************************************************************************
11   * declaration class TextNode
12   **************************************************************************/
13  class TextNode : public DocumentNode {
14    friend class MultiLayerDocumentParser;
15  public:
16
17    TextNode(DocumentItem *item);
18
```

```
19    virtual ~TextNode();
20
21    const itemid_t GetId() const;
22
23    DocumentNode::NodeType GetType() const;
24
25    wxString GetName() const;
26
27    wxString GetValue() const;
28
29    size_t GetChildrenCount() const;
30
31    DocumentNode* GetChild(const size_t idx);
32
33    void AppendChild(DocumentNode *node);
34
35    void InsertChild(DocumentNode *node, size_t idx);
36
37  private:
38    typedef std::vector<DocumentItem*> itemvector_t;
39    itemvector_t m_items;
40    mutable wxString m_value;
41    mutable bool m_value_valid;
42
43    void AddItem(DocumentItem *item);
44
45  }; // class TextNode
46
47
48  inline DocumentNode::NodeType TextNode::GetType() const {
49    return (DocumentNode::NODE_TEXT);
50  }
51
52
53  inline size_t TextNode::GetChildrenCount() const {
54    return (0);
55  }
56
57  #endif /* TEXTNODE_HH */
```

ElementNode.hpp

```
1   /* $id$
2    * Copyright (C) 2005 Oliver Schonefeld
3    * All rights reserved
4    */
5   #ifndef ELEMENTNODE_HH
6   #define ELEMENTNODE_HH
7   #include <vector>
8   #include "Document.hpp"
9
10  /*****************************************************************************
11   * declaration class ElementNode
12   *****************************************************************************/
13
14  class ElementNodeBase : public DocumentNode {
15    friend class MultiLayerDocumentParser;
16  public:
17    virtual ~ElementNodeBase();
18
19    const itemid_t GetId() const;
20
21    virtual DocumentNode::NodeType GetType() const = 0;
22
23    wxString GetName() const;
24
25    wxString GetValue() const;
26
27    virtual size_t GetChildrenCount() const = 0;
28
29    virtual DocumentNode* GetChild(const size_t idx) = 0;
30
31    virtual void AppendChild(DocumentNode *node) = 0;
32
33    virtual void InsertChild(DocumentNode *node, size_t idx) = 0;
34
35  private:
36    DocumentItem *m_item;
37
38  protected:
39
40    ElementNodeBase(DocumentItem *item);
41
42  }; // class ElementNode
43
```

```
44
45  inline const itemid_t ElementNodeBase::GetId() const {
46    return (m_item->GetId());
47  }
48
49
50  inline wxString ElementNodeBase::GetName() const {
51    return (m_item->GetName());
52  }
53
54
55  inline wxString ElementNodeBase::GetValue() const {
56    return (m_item->GetValue());
57  }
58
59
60  /*****************************************************************************
61   * declaration class ElementNode
62   *****************************************************************************/
63
64  class ElementNode : public ElementNodeBase {
65  public:
66    ElementNode(DocumentItem *item);
67
68    virtual ~ElementNode();
69
70    DocumentNode::NodeType GetType() const;
71
72    size_t GetChildrenCount() const;
73
74    DocumentNode* GetChild(const size_t idx);
75
76    void AppendChild(DocumentNode *node);
77
78    void InsertChild(DocumentNode *node, size_t idx);
79
80  private:
81    typedef std::vector<DocumentNode*> children_t;
82    children_t m_children;
83
84  }; // class ElementNode
85
86
87  inline DocumentNode::NodeType ElementNode::GetType() const {
88    return (DocumentNode::NODE_ELEMENT);
89  }
90
91  inline size_t ElementNode::GetChildrenCount() const {
92    return (m_children.size());
93  }
94
95  /*****************************************************************************
96   * declaration class EmptyElementNode
97   *****************************************************************************/
98
99  class EmptyElementNode : public ElementNodeBase {
100 public:
101   EmptyElementNode(DocumentItem *item);
102
103   virtual ~EmptyElementNode();
104
105   DocumentNode::NodeType GetType() const;
106
107   size_t GetChildrenCount() const;
108
109   DocumentNode* GetChild(const size_t idx);
110
111   void AppendChild(DocumentNode *node);
112
113   void InsertChild(DocumentNode *node, size_t idx);
114
115 }; // class EmptyElementNode
116
117
118 inline DocumentNode::NodeType EmptyElementNode::GetType() const {
119   return (DocumentNode::NODE_ELEMENT_EMPTY);
120 }
121
122 inline size_t EmptyElementNode::GetChildrenCount() const {
123   return (0);
124 }
125
126 #endif /* ELEMENTNODE_HH */
```

Attribute.hpp

```
1   /* $Id$
2    * Copyright (C) 2005 Oliver Schonefeld
3    * All rights reserved
4    */
5   #ifndef ATTRIBUTE_HH
6   #define ATTRIBUTE_HH
7   #include <vector>
8   #include <wx/string.h>
9
10
11  /*****************************************************************************
12   * definition class Attribute
13   *****************************************************************************/
14
15  class Attribute {
16  public:
17
18    explicit Attribute(const wxString name, const wxString value);
19
20    virtual ~Attribute();
21
22    wxString GetName() const;
23
24    void SetName(const wxString name);
25
26    wxString GetValue() const;
27
28    void SetValue(const wxString value);
29
30  private:
31    wxString m_name;
32    wxString m_value;
33  }; // class Attribute
34
35
36  inline wxString Attribute::GetName() const {
37    return (m_name);
38  }
39
40
41  inline wxString Attribute::GetValue() const {
42    return (m_value);
43  }
44
45
46  /*****************************************************************************
47   * definition class AttributeList
48   *****************************************************************************/
49
50  class AttributeList {
51  public:
52    explicit AttributeList();
53
54    virtual ~AttributeList();
55
56    size_t GetCount() const;
57  private:
58    typedef std::vector<Attribute*> AttributeVector;
59    AttributeVector m_attributes;
60  }; // class AttributeList
61
62
63  inline size_t AttributeList::GetCount() const {
64    return (m_attributes.size());
65  }
66
67  #endif /* ATTRIBUTE_HH */
```

DataAccess.hpp

```
1   /* $id$
2    * Copyright (C) 2005 Oliver Schonefeld
3    * All rights reserved
4    */
5   #ifndef DATAACCESS_HH
6   #define DATAACCESS_HH
7   #include <wx/string.h>
8
9
10  class DataAccess {
11  public:
12    typedef enum {
13      NORMAL,
```

```
14      FILTER
15    } Mode;
16
17    void SetMode(const Mode mode);
18
19    void SetParameter(const int offset, const int length);
20
21    wxString GetData(const int spos, const int epos) const;
22
23    wxString GetData(const int spos, const int epos,
24                     const Mode mode, const int offset,
25                     const int length) const;
26
27    bool IsOpeningTagAt(const int offset, const Mode mode,
28                        const int i_offset, const int i_length) const;
29
30    bool IsClosingTagAt(const int offset, const Mode mode,
31                        const int i_offset, const int i_length) const;
32
33    virtual int GetLength() const = 0;
34
35  protected:
36    DataAccess();
37
38    virtual wxString ReadRange(const int spos, const int epos) const = 0;
39
40  private:
41    Mode m_mode;
42    int m_offset;
43    int m_length;
44
45  }; // class DataAccess
46
47
48  inline wxString DataAccess::GetData(const int spos, const int epos) const {
49    return (GetData(spos, epos, m_mode, m_offset, m_length));
50  }
51
52  #endif /* DATAACCESS_HH */
```

## Erklärung

Hiermit erkläre ich, Oliver Schonefeld, dass die vorliegende Arbeit von mir alleine angefertigt wurde. Quellen und Sekundärliteratur sind in dem Literaturverzeichnis vollständig angegeben.

Oliver Schonefeld
Bielefeld, den 30. September 2005